GitforGits®

ASIAN PUBLISHING HOUSE

ПРОГРАММИРОВАНИЕ БЭКЕНДА НА Python

Практическое руководство

Тим Питерс

bhv®

PRACTICAL PYTHON BACKEND PROGRAMMING

Build Flask and FastAPI applications, asynchronous programming, containerization and deploy apps on cloud

Tim Peters

Тим Питерс

ПРОГРАММИРОВАНИЕ БЭКЕНДА НА Python

Практическое руководство

Санкт-Петербург
«БХВ-Петербург»
2025

УДК 004.43
ББК 32.973.26-018.1
П35

Питерс Т.

П35 Программирование бэкенда на Python. Практическое руководство: пер. с англ. — СПб.: БХВ-Петербург, 2025. — 288 с.: ил.

ISBN 978-5-9775-2096-6

Книга посвящена современным технологиям для программирования и поддержки серверной части (бэкенда) на Python. Рассказано о программировании на Python в облачной среде, управляемой через Docker и Kubernetes, о фреймворке Flask для веб-разработки на Python, о поглощении и преобразовании данных через FastAPI, об интеграции новых приложений и модулей Python с устоявшимися базами данных с применением SQLAlchemy, авторизации и аутентификации с применением OAuth, взаимодействии с брокерами сообщений Kafka и RabbitMQ, а также о повышении производительности языка Python и об эффективной работе с унаследованным кодом.

Для Python-разработчиков

УДК 004.43
ББК 32.973.26-018.1

Подписано в печать 02.09.25.
Формат 60×90 1/16. Печать офсетная. Усл. печ. л. 18.
Тираж 1000 экз. Заказ № 15377.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-81119177615 (англ.)
ISBN 978-5-9775-2096-6 (рус.)

Оглавление

Предисловие

Вы держите в руках небольшую и практичную книгу, описывающую основы разработки внутренних компонентов на Python. И новички, и самые разные специалисты, от Python-программистов до разработчиков, программирующих на других языках, программистов полного стека и веб-разработчиков, найдут в этой книге все, что им нужно знать, чтобы стать экспертами в программировании внутренних компонентов.

В книге рассматриваются ключевые темы разработки внутренних компонентов, включая создание стабильных сред разработки и использование виртуальных сред для более эффективного управления зависимостями. Изучив синтаксис, структуры данных и функции языка, читатели получат прочные знания о программировании на Python с акцентом на задачи внутренней части.

С помощью этой книги вы научитесь создавать и запускать динамические веб-приложения. Здесь подробно рассматриваются такие веб-фреймворки, как Flask и FastAPI. Кроме того, вы узнаете о SQLAlchemy, позволяющей эффективно работать с данными и интегрировать базы данных, а также здесь говорится о том, как улучшить работу приложений с базами данных PostgreSQL, MySQL и MongoDB. Стратегии управления одновременными операциями и повышения производительности также хорошо описываются. Кроме того, в книге рассматривается асинхронное программирование на Python.

Обеспечение безопасности имеет огромное значение при разработке внутренних систем. Здесь будет рассказано о различных методах аутентификации, безопасных протоколах связи, таких как HTTPS, и методах защиты REST API. С помощью брокеров

сообщений, таких как RabbitMQ и Kafka, вы сможете эффективно управлять асинхронными задачами и обрабатывать данные в реальном времени.

Благодаря этой книге читатели узнают, как упаковывать приложения в контейнеры и управлять ими с помощью технологий Docker и Kubernetes. Далее в книге рассказывается о том, как использовать бессерверные архитектуры, как применять современные инструменты для непрерывной интеграции и развертывания, а также как развертывать приложения на облачных платформах, таких как AWS.

Эта книга претендует на то, чтобы научить читателей писать для внутренних систем надежный, эффективный и читабельный код на Python, а также строить, развертывать и поддерживать такие системы.

Из этой книги вы узнаете, как:

- Создавать динамические веб-приложения с продуманной логикой внутренней части с использованием Flask и FastAPI.
- Написать эффективный, хорошо структурированный код внутренней части приложения, изучив синтаксис, функции и лучшие практики Python.
- Сделать с помощью методов асинхронного программирования свои приложения более эффективными и масштабируемыми.
- Изучить Kubernetes и Docker для автоматизации и контейнеризации приложений, чтобы улучшить их развертывание и масштабируемость.
- Использовать облачные сервисы AWS для развертывания приложений с гарантированным временем безотказной работы и молниеносной производительностью.
- Повысить эффективность и совместимость, создавая и управляя средами разработки на Python.
- Усовершенствовать свои возможности по работе с данными, научившись интегрировать базы данных и манипулировать ими с помощью SQLAlchemy.

- Защищать свои онлайн-приложения с помощью сложных функций авторизации и аутентификации OAuth и JWT.
- Эффективно обрабатывать данные в режиме реального времени и передавать сообщения с помощью RabbitMQ и Kafka.
- Оптимизировать процессы, сократить количество ошибок и внедрить непрерывную интеграцию и развертывание, следуя передовому опыту.

Пролог

Добро пожаловать в мир программирования внутренних компонентов приложений на Python. Я — Тим Питерс — в своей книге использовал свой многолетний опыт разработки программного обеспечения, чтобы помочь вам сориентироваться в этой сложной теме. Если вы опытный разработчик Python, программист, не изучающий Python, но пытающийся освоить его просторы, разработчик полного стека, желающий улучшить свои навыки работы с внутренними компонентами, или начинающий веб-разработчик, то эта книга для вас.

Решение написать эту книгу я принял, столкнувшись с трудностями при переходе от написания простых скриптов к разработке надежных внутренних систем. За годы работы я понял, что архитектура внутренней части — это фундамент любого успешного приложения. Обработка данных, обеспечение безопасности и бизнес-логика — все это происходит именно здесь. Так как язык программирования Python набрал большую популярность и стал основным языком для разработки внутренней архитектуры, я увидел необходимость в создании пособия, которое поможет разработчикам пройти через все этапы этого процесса, которые уже были пройдены и тщательно изучены мною.

Эта книга начинается с изучения основ создания среды разработки на языке Python, оптимизированной для создания внутренних компонентов. Вы узнаете, как с помощью соответствующих инструментов и библиотек настроить Python, чтобы процесс кодирования стал более совершенным. После этого мы погрузимся в самое сердце программирования на Python, где я помогу вам освоить синтаксис, структуры данных и фундаментальные концепции. Это даст вам прочный фундамент для дальнейшего развития.

Затем мы перейдем к более сложным темам, таким как веб-разработка с использованием таких популярных фреймворков, как Flask и FastAPI. В этих главах будет рассказано не только о том, как создавать приложения. Здесь речь пойдет о структуре и шаблонах проектирования, которые приводят к созданию масштабируемого и поддерживаемого кода.

Поскольку данные являются фундаментом программирования внутренних компонентов, эта книга посвящена технологиям управления базами данных. Вы будете работать с реляционными базами данных, такими как PostgreSQL и MySQL. Также я расскажу о системах управления неструктурированных баз данных NoSQL, например MongoDB, чтобы понять, как эти базы эффективно использовать. На примере программного обеспечения SQLAlchemy мы увидим, как интеграция ORM может упростить работу с базами данных в ваших приложениях.

По мере развития приложений этим приложениям потребуется решать все более сложные задачи. Например, асинхронные операции и обработка данных в реальном времени. Я покажу вам, как выполнять асинхронное программирование на Python и обрабатывать данные в реальном времени с помощью Kafka. Вы узнаете, как управлять фоновыми задачами с помощью Celery и создавать масштабируемые приложения, способные без проблем обрабатывать большие объемы данных.

Безопасность приложений и данных очень важна. И в этой книге вы найдете исчерпывающие сведения о том, как защитить свои приложения, — от аутентификации JWT и OAuth до создания безопасных конечных точек API и внедрения SSL.

Наконец, ни одна книга о современных внутренних компонентах не будет полной без рассмотрения вопросов развертывания и масштабируемости. Вы узнаете, как, используя Docker, выполнять контейнерную обработку приложений, управлять ими с помощью Kubernetes и разворачивать их в облаке AWS.

Данная книга — это не просто руководство по программированию, это путь к тому, чтобы стать опытным разработчиком внутренних компонентов, умеющим создавать и внедрять мощные и эффективные приложения. Присоединяйтесь ко мне в этом путешествии и давайте творить невероятные вещи с использованием Python.

Предварительные требования

Начальные навыки

Независимо от того, являетесь ли вы опытным разработчиком, программирующем на языке Python, программистом, не знакомым с этим языком программирования, тем, кто только начинает осваивать программирование на Python, разработчиком полного стека, стремящимся усовершенствовать свои навыки работы с внутренними компонентами, или начинающим веб-разработчиком, делающим первые шаги в программировании, эта книга создана для вас.

Использование кодов

Вам нужны примеры кода, которые помогут вам в разработке программ или написании документации? Не сомневайтесь! В нашей книге вы найдете множество дополнительных материалов, включая примеры кодов и упражнения.

Эта книга не только поможет вам в работе, но и позволит воспользоваться примерами кода в ваших программах и документации. Однако обратите внимание, что если вы воспроизводите значительную часть кода, мы просим вас связаться с нами для получения разрешения.

Но не беспокойтесь: использование в вашей программе нескольких небольших фрагментов кода из этой книги или воспроизведение примера кода при ответе на вопрос, но со ссылкой на нашу книгу, разрешения не требует. Но если вы все же решите отдать должное, как правило, в ссылке на авторство следует указать ав-

тора, название книги, издательство и ISBN. Например: Tim Peters «Practical Python Backend Programming» (Тим Питерс «Программирование серверной части на Python. Практическое руководство»).

Если у вас есть сомнения в том, подпадает ли предполагаемый пример кода, который вы хотели бы использовать, под условия добросовестного использования или требует разрешения, как указано выше, пожалуйста, свяжитесь с нами по адресу:

mail@bhv.ru.

Мы будем рады помочь и прояснить любые вопросы.

Благодарности

Я в огромном долгу перед командой GitforGits за их неослабевающий энтузиазм и мудрые советы, которые они давали на протяжении всего процесса написания этой книги. Их знания и тщательное редактирование помогли убедиться в том, что эта книга будет полезна людям с любым уровнем подготовки и навыками работы. Кроме того, я хотел бы поблагодарить всех, кто участвовал в издательском процессе, благодаря их усилиям проект стал тем, чем он является сегодня.

Наконец, я хотел бы выразить свою благодарность всем, кто на протяжении всей моей жизни проявлял ко мне безусловную любовь и поддержку. Их поддержка сыграла решающую роль в завершении работы над этой книгой. Я ценю вашу помощь в этом начинании и ваш постоянный интерес к моей карьере.

ГЛАВА 1
Основы разработки внутренних компонентов

Введение

В первой главе мы познакомимся с важнейшими компонентами, составляющими основу современных веб-приложений. Здесь закладывается фундамент для понимания того, что такое разработка внутренних компонентов (так называемое *бэкенд-программирование*) и какую ключевую роль играет язык Python в этом процессе. Мы начнем с определения разработки внутренних компонентов, подробно описывая функциональные возможности и процессы, происходящие за кулисами веб-приложения.

Затем обсудим, почему именно язык Python считается самым популярным для бэкенд-программирования, узнаем о простоте этого языка, гибкости, а также о надежной экосистеме фреймворков и библиотек, которые поддерживают данный язык. Чтобы предоставить вам необходимые инструменты, мы расскажем о настройке среды разработки, в которую входят Python, Visual Studio Code и Linux. Такая настройка гарантирует, что у вас будет надежная и согласованная платформа для разработки внутренних компонентов приложений.

Затем мы познакомимся с концепцией виртуальных сред, которые необходимы для управления зависимостями и предотвращения конфликтов между проектами. Поскольку навыки работы с командной строкой крайне важны для разработчиков внутренних компонентов, мы рассмотрим основы интерфейса командной строки (CLI) и расскажем вам о том, как с помощью этого инструмента эффективно управлять системой.

Контроль версий — еще один краеугольный камень в профессиональном развитии программиста. Поэтому мы познакомим вас

с Git, самой распространенной системой для отслеживания внесенных изменений и совместной работы в программных проектах. Чтобы освежить и закрепить ваши навыки работы с Python, мы изучим такие фундаментальные понятия, как синтаксис, структуры данных и функции, чтобы вы были хорошо подготовлены к написанию кода.

И в завершение главы мы расскажем о передовом опыте программирования на Python. Эти рекомендации помогут вам писать чистый, поддерживаемый и эффективный код, формируя хорошие привычки на ранних этапах вашей карьеры разработчика внутренних компонентов. Задача этой главы — создание прочного фундамента, который позволит вам строить более сложные и динамичные внутренние системы в последующих главах.

Описание разработки внутренних компонентов

Разработка внутренних систем веб-приложения, которые часто называют серверной частью, — это основная составляющая веб-разработки, которая связана с логикой, взаимодействием с базой данных, аутентификацией пользователей и конфигурацией сервера. Работа внутренних компонентов не видна пользователям, но очень важна для функционирования приложения. Сейчас мы рассмотрим основные составляющие внутренних частей веб-приложения и их роль в обычной веб-среде.

Основные внутренние компоненты

Сервер

В рамках веб-приложений под термином «сервер» можно понимать как программное, так и аппаратное обеспечение. С точки зрения аппаратного обеспечения сервер — это компьютер, предназначенный для обработки запросов и доставки данных на другие компьютеры через локальную сеть или Интернет. По сравнению с обычными компьютерами серверное оборудование предназначено для выполнения более сложных задач, обладает

увеличенным объемом памяти и эффективно управляет сетевым трафиком. Кроме того, серверы должны обладать очень высокой надежностью, доступностью и возможностью масштабирования, справляться с переменной нагрузкой и работать непрерывно и без сбоев.

С точки зрения программного обеспечения сервер — это приложение или служба, работающая на серверном аппаратном обеспечении и ожидающая обработки запросов от клиентов. К этому программному обеспечению относятся веб-серверы, почтовые серверы, файловые серверы и другие. Наиболее распространенным типом сервера в веб-разработке является веб-сервер.

Веб-серверы предназначены для хранения, обработки и доставки веб-страниц пользователям. Они обрабатывают HTTP-запросы, поступающие от клиентов. Эти запросы обычно создаются с помощью браузеров или других веб-приложений.

Ниже приведено пошаговое описание их работы.

1. Сервер прослушивает запросы, отправленные через Интернет на определенный IP-адрес и порт. Получив запрос, сервер считывает и интерпретирует HTTP-заголовки, чтобы определить характер запроса.
2. В зависимости от условий запроса (например, получение веб-страницы, запрос к базе данных или отправка данных формы) сервер выполняет требуемые операции. При этом могут выполняться сценарии на стороне сервера или запросы к базам данных для получения или обновления информации.
3. Веб-серверам часто приходится загружать ресурсы, из которых создаются веб-страницы, включая файлы HTML, CSS, используемые для оформления внешнего вида веб-страницы, файлы JavaScript и медиафайлы. Эти файлы часто хранятся на сервере и должны быть быстро найдены, чтобы обеспечить незамедлительное реагирование на запросы пользователей.
4. Для динамических страниц, требующих внутренней обработки (например, PHP, сценарии Python или Java), сервер выполняет эти сценарии, генерируя HTML в зависимости от текущего состояния базы данных или бизнес-логики сервера.

5. После обработки запроса и получения всех необходимых данных сервер упаковывает все в HTTP-ответ и отправляет его обратно клиенту. Этим ответом может быть полностью подготовленная HTML-страница, набор данных JSON/XML для API или сообщение о состоянии запроса.

База данных

База данных — это организованная коллекция данных, которые не только хранятся, но и доступны в электронном виде. И доступ к этим данным происходит с помощью компьютерной системы. Базы данных предназначены для обеспечения эффективного, надежного и удобного способа хранения и получения данных различных типов. Система управления базами данных (СУБД) служит интерфейсом между базой данных и конечными пользователями или прикладными программами, обеспечивая организацию и сопровождение данных без избыточности и несогласованности.

Ниже представлены основные функции баз данных для внутренних компонентов.

- Базы данных представляют собой систематизированный и организованный способ хранения информации, позволяющий эффективно ее извлекать и обрабатывать. Организация данных включает в себя категоризацию и индексацию. Индексация способствует быстрому поиску и извлечению информации.

  Примечание переводчика

  Категоризация данных — это процесс распределения различных данных по категориям (или тематическим группам) на основе смысловой близости.

  Индексация — это метод оптимизации базы данных, когда в базе данных создается дополнительная структура, называемая индексом. В ней хранятся ссылки на записи в таблице. Данная структура помогает организовать и сортировать данные таким образом, чтобы приложение могло более эффективно получать к ним доступ и быстрее выполнять запросы. Индекс обеспечивает ускоренный поиск данных, сокращая время выполнения запроса.

- С помощью запросов базы данных позволяют получать информацию в динамическом режиме. Это очень важно для об-

служивания пользовательских запросов, проведения статистического анализа и создания отчетов.

> ***Примечание переводчика***
>
> *Динамический режим в базе данных MS SQL Server* — это режим, при котором сервер под свои нужды занимает всю доступную память. Если операционной системе или другому приложению, работающему на этом же сервере, требуется больше памяти, то операционная система обрабатывает соответствующий запрос и MS SQL Server освобождает необходимое количество памяти. Динамический режим назначается по умолчанию при настройке сервера как универсальный режим.

- Базы данных обеспечивают возможность вставки, изменения и удаления хранящихся в базах определенных данных. Это необходимо, чтобы данные не устаревали и всегда соответствовали данному моменту времени, взаимодействуя с приложением в реальном времени.
- Многие базы данных поддерживают свойства ACID (Atomicity, Consistency, Isolation, Durability или Атомарность, Согласованность, Изоляция, Долговечность), которые позволяют обеспечить надежную обработку всех транзакций. Это означает, что база данных гарантирует успешное завершение всех операций в рамках транзакции перед ее фиксацией. При возникновении ошибки транзакция прерывается, а в базе данных все остается без изменений.
- Базы данных позволяют одновременно нескольким пользователям получать доступ к данным и работать с ними без помех, используя блокировки или предоставляя несколько версий данных.

Интерфейсы прикладного программирования (API)

API можно представить как договор между поставщиком и пользователем услуги. Провайдер обязуется предоставлять сервис в соответствии со спецификациями, изложенными в документации API, а пользователь — взаимодействовать с сервисом в строгом соответствии с этими спецификациями. API реализуются с помощью вызовов функций, состоящих из глаголов и сущест-

вительных. Существительные — это ресурсы, а глаголы — действия, которые должны быть выполнены над этими ресурсами.

Ниже приведены типы API для веб-разработки.

- **Веб-интерфейсы API.** Работают как на серверной, так на клиентской стороне и обеспечивают взаимодействие через Интернет. Данное взаимодействие обычно осуществляется по протоколу HTTP, что упрощает совместную синхронную работу веб-систем. Наиболее популярными типами веб-интерфейсов являются REST и GraphQL.
- **Внутренние API.** Используются внутри организации для интеграции различных внутренних систем и сервисов. Действие этих API не распространяется за пределы организации, обеспечивая дополнительный уровень безопасности и позволяя создавать индивидуальные функции, отвечающие потребностям бизнеса.
- **Внешние API.** Доступны для внешних пользователей, позволяя сторонним разработчикам получать доступ к функциям или ресурсам инструмента или сервиса. Такие API требуют тщательного документирования и повышенных мер безопасности, чтобы обеспечить управление взаимодействием внешней стороны с системой.
- **API сторонних разработчиков.** Эти API предоставляются третьими сторонами и обычно используются для обработки платежей, взаимодействия с социальными сетями или передачи данных (например, API, предоставляемые PayPal, Twitter или Google Maps).

Веб-фреймворк

Веб-фреймворк — это основа программного обеспечения для разработки веб-приложений. В его состав входят веб-сервисы, веб-ресурсы и веб-интерфейсы. Фреймворки содержат библиотеки и инструменты, помогающие разработчикам создавать приложения быстрее и эффективнее за счет автоматизации стандартных процессов.

Большинство веб-фреймворков предлагают несколько ключевых компонентов.

- **Маршрутизация.** Служит для определения маршрутов, по которым входящие HTTP-запросы направляются к соответствующим обработчикам. Разработчики определяют маршруты с помощью шаблонов и указывают, какие функции будут вызываться в зависимости от входящего URL, в том числе с поддержкой динамических параметров URL.
- **Объекты запросов и ответов.** Фреймворки упрощают работу с HTTP-запросами и ответами, преобразуя их в объекты. Таким образом, разработчики получают простой интерфейс для доступа к данным и управления типом ответа, возвращаемого клиенту.
- **Движок шаблонов.** Движки шаблонов позволяют разработчикам создавать динамические HTML-страницы, встраивая логику внутренней части (например, циклы, условия и переменные данные) в HTML. Например, так происходит разделение дизайна и кода, что делает веб-приложения более удобными в управлении и модификации.
- **Интеграция баз данных.** Веб-фреймворки часто комплектуются встроенными или дополнительными инструментами ORM (Object-Relational Mapping или объектно-реляционное отображение), которые абстрагируют работу с базами данных. Это позволяет разработчикам взаимодействовать с базами данных, используя конструкции языка программирования, а не создавать SQL-запросы напрямую (т. е. работать с базами данных так, как если бы это были объекты из языков программирования).
- **Поддержка Middleware.** Middleware — это механизмы или сервисы, которые выполняются до или после обработки запроса. Эти механизмы полезны для реализации функций, которые должны обрабатываться в каждом запросе. Например, аутентификация пользователей, обеззараживание данных и управление сеансами.
- **Функции безопасности.** Фреймворки обеспечивают средства безопасности для защиты веб-приложений от таких распространенных уязвимостей, как SQL-инъекции, межсайтовый скриптинг (XSS) и подделка межсайтовых запросов (CSRF).

Они справляются с этими угрозами с помощью различных форм санации ввода, безопасных настроек по умолчанию и другого передового опыта.

Middleware (связующее или промежуточное программное обеспечение)

Middleware — это программное обеспечение, которое находится между приложением и сетью или системой, на которой оно работает. С его помощью осуществляется связь между различными компонентами приложения или разными приложениями в целом, что облегчает взаимодействие и управление данными. Услуги связующего программного обеспечения могут состоять из служб обмена сообщениями, аутентификации, управления контентом, управления API и т. д.

Часто связующее программное обеспечение рассматривается в рамках определенных фреймворков, в которых является центральной частью архитектуры.

Ниже приведены несколько примеров.

- **Express.js.** В фреймворке Express.js (Node.js) функции связующего программного обеспечения — это функции, которые имеют доступ к объекту запроса (req), объекту ответа (res) и следующей связующей функции в цикле «запрос-ответ» приложения. Данные функции могут выполнять любой код, вносить изменения в объекты запроса и ответа, завершать цикл «запрос-ответ» и вызывать следующую промежуточную функцию.
- **Django.** В Django (Python) связующее программное обеспечение — это небольшой плагин, который обрабатывает запросы и ответы. Django поддерживает список связующего программного обеспечения для выполнения запросов и ответов, и каждый компонент должен быть разработан таким образом, чтобы быть совместимым с другими промежуточными программами в цепочке.
- **ASP.NET Core.** Здесь компоненты связующего программного обеспечения собираются в конвейер приложений для обработки запросов и ответов. Каждый компонент может выполнять

операции как до, так и после следующего компонента в конвейере.

Кеширование

Кеширование — это процесс хранения данных в кеше, высокоскоростном слое хранения данных, чтобы последующие запросы на получение именно этих данных выполнялись быстрее. Данные, хранящиеся в кеше, могут быть результатом предыдущих вычислений или дубликатом данных, хранящихся в другом месте. Когда поступает запрос на данные, находящиеся в кеше, система может обойти более медленные внутренние процессы и быстро предоставить ответ.

Общий процесс кеширования включает в себя несколько этапов:

1. Пользователь или клиентское приложение делают запрос на получение данных.
2. Система проверяет, есть ли запрашиваемые данные в кеше.
3. Если данные находятся в кеше («попадание в кеш»), они возвращаются немедленно, минуя более медленные процессы получения данных. Если данные не содержатся в кеше («пропуск кеша»), то запрос обрабатывается в обычном режиме и данные извлекаются из более медленного внутреннего интерфейса или системы хранения.

После того как данные получены, они сохраняются в кеше, чтобы последующие запросы на те же данные выполнялись быстрее.

Чтобы рационально использовать память и не допускать устаревания информации в кеше, старые или редко используемые данные периодически из кеша удаляются. Для этого часто используются алгоритмы типа Least Recently Used (LRU, редко используемые) или First In First Out (FIFO, первым пришел, первым ушел).

Функционирование внутренних компонентов

При взаимодействии пользователя с внешним приложением, например при открытии ссылки или заполнении формы, запрос отправляется на сервер через Интернет.

Ниже описано, как все вышеперечисленные компоненты взаимодействуют друг с другом.

1. **Обработка запросов.** Сервер, на котором установлен веб-фреймворк, получает запрос. Веб-фреймворк направляет запрос в соответствующий контроллер на определенный URL, соответствующий типу запроса (GET, POST и т. д.).
2. **Процесс обработки с помощью связующего программного обеспечения.** Прежде чем запрос достигнет контроллера, он может пройти через различное связующее программное обеспечение. Здесь могут решаться такие задачи, как регистрация, аутентификация пользователя и проверка данных. Связующее программное обеспечение может отклонять запросы, не требуя их дальнейшей обработки, или передавать их далее в систему.
3. **Выполнение бизнес-логики.** Как только запрос доходит до контроллера, сервер запускает выполнение бизнес-логики, в ходе чего может быть или выполнен запрос, или произойти обновление базы данных через ORM (Object Relational Mapping или объектно-реляционное отображение) или напрямую с помощью SQL-запросов.
4. **Взаимодействие с API.** Если приложение использует для работы внешние сервисы, сервер, чтобы получить данные или выполнить операции, может обращаться к внешним системам через API. Такое взаимодействие часто носит асинхронный характер и требует тщательной обработки для поддержания требуемого уровня производительности.
5. **Ответ на данные.** После обработки данных сервер подготавливает ответ, который может включать извлеченные данные из кеша, что позволяет сократить время на подготовку ответа. Ответ отправляется обратно клиенту в формате HTML, JSON или XML.
6. **Клиентское обновление.** Клиент (браузер или мобильное приложение) получает ответ и соответствующим образом обновляет пользовательский интерфейс. Если ответ представляет собой данные (обычно в формате JSON), внешний интерфейс использует их для обновления состояния приложения или отображает (render) их по мере необходимости.

Разработка внутренних компонентов — сложный, ответственный и критически важный этап всего процесса веб-разработки, требующий глубокого понимания серверных технологий, управления базами данных, связующего программного обеспечения, разработки API и механизмов кеширования. Владение этими элементами обеспечивает создание надежных, эффективных и масштабируемых веб-приложений.

Роль Python в разработке внутренних компонентов

Благодаря своей простоте, универсальности и надежной экосистеме Python занял лидирующие позиции в разработке внутренних компонентов. Главная особенность Python в том, что с его помощью можно быстро создать приложение, решить задачи, связанные с внутренними компонентами, и интегрироваться с различными системами и технологиями. Благодаря такой универсальности Python становится основным выбором для разработчиков, стремящихся создать надежные, масштабируемые и обслуживаемые внутренние системы.

Универсальность и читабельность

Синтаксис Python ясен и лаконичен, благодаря чему этот язык является прекрасным инструментом для разработки внутренних компонентов. Благодаря его читабельности даже сложные системы легко понять и поддерживать, что снижает вероятность ошибок и упрощает процесс отладки. Такая простота использования распространяется и на написание и развертывание кода на серверной стороне, где простой синтаксис Python позволяет разработчикам формулировать концепции без создания дополнительного кода. Например, использование пробелов и общих выражений в Python позволяет разработчикам сосредоточиться на логике, а не на формальностях синтаксиса, что ускоряет процесс разработки.

Универсальная стандартная библиотека

Универсальная стандартная библиотека Python — одно из главных достоинств этого языка. В состав даной библиотеки входят модули и пакеты для решения практически любых задач программирования. Здесь и доступ к базам данных, и работа с файлами, и взаимодействие между процессами, и даже интернет-протоколы, такие как HTTP и FTP. Такой широкий спектр поддерживаемых операций делает Python универсальным инструментом для разработчиков внутренних компонентов, которые могут использовать стандартную библиотеку Python для эффективного выполнения множества задач внутренних частей приложения без необходимости использования сторонних модулей.

Фреймворки и инструменты для разработки внутренних компонентов

Экосистема Python включает в себя несколько мощных фреймворков и инструментов, специально предназначенных для разработки внутренних компонентов, которые помогают быстро создавать надежные веб-приложения.

Перечислим популярные фреймворки.

- **Django.** Известный своим подходом «батарейки в комплекте», Django оснащен множеством встроенных функций, таких как ORM, механизмы аутентификации, средства передачи сообщений. Все эти инструменты крайне необходимы для разработки современных веб-приложений. Структура Django способствует быстрой разработке высококачественных веб-приложений.
- **Flask.** Микрофреймворк Flask содержит все самое необходимое для запуска веб-приложения. Его простота и гибкость позволяют использовать этот инструмент для разработки небольших и средних по объему приложений, также он подходит разработчикам, которые предпочитают добавлять расширения в зависимости от конкретных требований проекта.
- **FastAPI.** Это современный, быстрый (высокопроизводительный) веб-фреймворк для создания API на основе Python 3.7+

со стандартными аннотациями типов Python. Ключевыми особенностями FastAPI считаются быстрота выполнения кода и меньшее количество ошибок.

Примечание редактора

Type hints Python — аннотации (или подсказки) типов, позволяющие указывать ожидаемые типы данных для переменных, функций и классов, используемых в коде на Python, для которого характерна динамическая типизация. FastAPI применяет аннотации типов Python для обработки запросов, что обеспечивает проверку данных и автоматическое документирование API.

Поддержка асинхронной работы

В Python 3.5 добавлена встроенная поддержка асинхронного программирования с помощью библиотеки Asyncio, которая обеспечивает асинхронные операции ввода-вывода. Асинхронная поддержка критически важна при обработке больших объемов данных и запросов, что часто и происходит в современных веб-приложениях. Эта технология особенно эффективна для улучшения производительности и быстродействия приложений, связанных с операциями ввода-вывода и сетью. Такие фреймворки, как FastAPI и Sanic, разработаны с учетом всех преимуществ асинхронных возможностей Python, позволяя разработчикам обрабатывать больше соединений и данных с использованием меньшего количества оборудования.

Возможность интеграции

Python выгодно отличается своей способностью интегрироваться с другими языками и технологиями. Это очень важно при разработке внутренних компонентов, в которых приложениям часто приходится взаимодействовать с унаследованными системами, использовать сторонние сервисы и выполнять анализ данных. Совместимость Python с языком C/C++ и возможность работы практически на всех операционных системах делают его универсальным выбором для разработки внутренних систем. Кроме того, многочисленные библиотеки и API Python для взаимодействия с данными позволяют легко интегрировать сложный функционал в любое приложение.

Возможность подключения к базам данных

Python работает с различными базами данных, от традиционных реляционных баз, таких как MySQL и PostgreSQL, до современных баз данных NoSQL, таких как MongoDB и Cassandra. Библиотеки, такие как SQLAlchemy и ORM Django, позволяют эффективно управлять базами данных и выполнять операции с ними, абстрагируя многие операции с базами данных и взаимодействие с ними, что делает интеграцию баз данных более простой и надежной.

Сообщество и ресурсы

Python пользуется огромной популярностью благодаря поддержке активного сообщества разработчиков, которые вносят свой вклад в создание огромного количества библиотек и фреймворков. Это сообщество не только стимулирует развитие языка, но и оказывает огромную поддержку с помощью документации, форумов, учебных пособий и пакетов сторонних разработчиков. Независимо от того, сталкивается ли разработчик с общей проблемой или ищет специализированную библиотеку, вероятность того, что такой пакет для Python уже существует, очень высока.

Искусственный интеллект и машинное обучение

Python также является основным языком, используемым в области искусственного интеллекта (ИИ) и машинного обучения (МО) — областях, которые становятся наиболее актуальными при разработке внутренних компонентов. Особенно в сценариях, связанных с обработкой данных, прогностическим анализом и автоматическим принятием решений. Такие библиотеки, как TensorFlow, PyTorch и Scikit-learn, широко используются во внутренних системах для анализа пользовательских данных и совершенствования динамических процессов принятия решений.

Роль Python в разработке внутренних компонентов очень большая благодаря его простоте, широкому набору фреймворков и библиотек, возможностям асинхронной работы, поддержке баз данных и мощной поддержке сообщества. Все эти характеристи-

ки позволяют Python стать адаптируемым, мощным и эффективным инструментом для разработчиков, создающих основу современных веб-приложений.

Настройка среды разработки: Python, VS Code и Linux

Установка Linux

Если вы решили создать новую среду разработки, то первым шагом будет установка операционной системы Linux. Ubuntu — это популярный дистрибутив операционной системы Linux, который отличается широкой поддержкой и удобством использования, что делает его отличным выбором для разработчиков.

1. Посетите официальный сайт Ubuntu и загрузите последнюю версию LTS (Long Term Support, долгосрочная поддержка).
2. Создайте загрузочный USB-накопитель с ISO-файлом Ubuntu с помощью таких инструментов, как Rufus или Etcher.
3. Вставьте USB-накопитель в разъем компьютера, перезагрузитесь и войдите в настройки BIOS. Измените порядок загрузки на запуск с USB-накопителя, сохраните изменения и выйдите. Следуйте инструкциям по установке Ubuntu, выбрав язык, раскладку клавиатуры и разметку диска.

Установка Python

Операционная система Ubuntu содержит предварительно установленный язык программирования Python. Однако его версия может оказаться устаревшей. Чтобы установить последнюю версию Python, воспользуйтесь терминалом:

1. Нажмите комбинацию клавиш <Ctrl>+<Alt>+<T>, чтобы запустить окно терминала.
2. Чтобы обновить список пакетов, выполните команду

```
sudo apt update
```

3. Установите Python, выполнив команду

```
sudo apt install python3
```

4. Проверьте установленную версию, набрав команду

```
python3 --version
```

Установка Visual Studio Code (VS Code)

VS Code — это легкий, но мощный редактор исходного кода, который поддерживает разработку на языке Python и может быть дополнен расширениями.

Загрузите VS Code:

1. Перейдите на официальный сайт Visual Studio Code.
2. Загрузите пакет **.deb** (для дистрибутивов Debian/Ubuntu).

Установите VS Code:

1. Запустите терминал.
2. Перейдите в каталог, в который был загружен файл VS Code **.deb**.
3. Установите пакет с помощью команды

```
sudo dpkg -i code_*.deb
```

(замените символ * на номер версии).

4. Устраните зависимости (если они есть), выполнив команду

```
sudo apt -f install
```

5. Вы можете запустить VS Code из меню приложений или просто набрав команду `code` в терминале.

Настройка VS Code для Python

Чтобы использовать язык Python для разработки приложений с помощью редактора VS Code, настройте его с помощью необходимых расширений и параметров.

Установите расширение Python для VS Code:

1. Запустите VS Code.
2. Перейдите к просмотру расширений, нажав значок в виде квадрата на боковой панели или воспользовавшись комбинацией клавиш <Ctrl>+<Shift>+<X>.

3. Найдите «Python» и установите расширение, предоставленное Microsoft.

Настройте интерпретатор Python:

1. Откройте файл Python или создайте новый.
2. Нажмите комбинацию клавиш <Ctrl>+<Shift>+<P>, чтобы открыть палитру команд.
3. Выберите **Python | Select Interpreter** и соответствующую версию Python, которая установлена в вашей системе.

Настройка виртуальной среды

Виртуальные среды в Python позволяют управлять зависимостями отдельно для каждого проекта. Вы можете создать виртуальную среду, выполнив следующие действия:

1. Установите virtualenv, выполнив команду:

   ```
   pip install virtualenv
   ```

2. Создайте виртуальную среду:

   - откройте в терминале каталог вашего проекта;
   - выполните команду

     ```
     virtualenv venv
     ```

     где `venv` — имя виртуальной среды.

3. Активируйте виртуальную среду в Linux, выполнив команду

   ```
   source venv/bin/activate
   ```

   Вид вашего приглашения изменится, и это значит, что вы работаете в `venv`.

4. По завершении работы выйдите из виртуальной среды, выполнив команду `deactivate`.

Заключительные этапы и тестирование

Теперь, когда вы создали среду разработки на основе Linux, Python и VS Code, пришло время эту среду протестировать, создав простое Python-приложение.

Создайте тестовый файл:

1. В VS Code создайте новый файл с именем **test.py**.
2. Напишите простой сценарий на языке Python, например:

```
print("Hello, World!")
```

Запустите тестовый файл:

1. Откройте терминал в VS Code, нажав сочетание клавиш <Ctrl>+<’> (символ обратного апострофа).
2. Убедитесь, что ваша виртуальная среда активирована.
3. Запустите скрипт, набрав команду

```
python test.p
```

В терминале должно появиться сообщение «Hello, World!», что свидетельствует о правильной настройке Python и его работоспособности. Теперь эта среда готова к выполнению более сложных задач по разработке внутренних компонентов, включая разработку веб-приложений с помощью таких фреймворков, как Django или Flask.

Знакомство с виртуальными средами

Что такое виртуальная среда?

Виртуальная среда — это изолированная рабочая область, которая позволяет устанавливать библиотеки Python в каталог, не влияя на общую установку Python. Виртуальная среда особенно полезна, когда для разных проектов требуются разные версии одного и того же пакета. Использование виртуальных сред позволяет избежать конфликтов между версиями. Каждая виртуальная среда имеет свой собственный двоичный файл Python (который соответствует версии интерпретатора Python, использованного для ее создания) и может иметь свой собственный независимый набор установленных пакетов Python в директориях сайта.

Зачем нужна виртуальная среда?

Перечислим основные факторы для использования виртуальных сред при разработке на Python.

- Виртуальные среды позволяют управлять зависимостями, которые требуются для разных проектов, создавая для этих проектов изолированные пространства. Это означает, что вы можете использовать разные версии библиотеки для разных проектов, не допуская конфликтов.
- Использование виртуальной среды гарантирует, что у всех разработчиков, которые работают над проектом, одинаковые зависимости. А это уменьшает проблемы типа «на моей машине работает».
- Благодаря тому, что все зависимости проекта изолированы в виртуальной среде, становится проще понять, что именно должно быть развернуто вместе с приложением. А это, в свою очередь, снижает вероятность потери модулей.
- Виртуальные среды позволяют экспериментировать с различными пакетами и обновлять их версии без риска повлиять на другие разработки.

Как настроить и использовать виртуальную среду?

Для настройки виртуальной среды необходимо выполнить несколько простых шагов.

Установка

Модуль `venv` — это стандартный инструмент для создания виртуальных сред в Python 3, и он уже предварительно установлен. Для Python 2 модуль `virtualenv` необходимо устанавливать отдельно.

Создание виртуальной среды

1. Если вы используете Python 3, перейдите в каталог проекта в терминале и выполните команду:

   ```
   python3 -m venv myenv
   ```

   заменив слово *myenv* на имя вашей виртуальной среды.

2. Для Python 2 после установки `virtualenv` следует использовать команду:

```
virtualenv myenv
```

Активация виртуальной среды

В Linux или macOS активируйте виртуальную среду, выполнив команду:

```
source myenv/bin/activate
```

Обычно после активации в приглашении терминала появляется имя виртуальной среды, заключенное в круглые скобки. Это означает, что любые команды Python или pip будут воздействовать только на эту среду.

Установка пакетов

Когда среда активирована, вы можете устанавливать пакеты, как обычно, с помощью инструмента pip:

```
pip install package_name
```

где *package_name* — имя пакета.

Деактивация

Для остановки виртуальной среды и возврата к глобальной среде Python просто выполните команду:

```
deactivate
```

Управление зависимостями

Чтобы отслеживать зависимости проекта, создайте файл **requirements.txt**, выполнив команду:

```
pip freeze > requirements.txt
```

Этот файл позволяет установить все необходимые пакеты в другой среде или на другой машине:

```
pip install -r requirements.txt
```

Передовой опыт

- Создайте для каждого проекта отдельную виртуальную среду, чтобы разделить зависимости, которые нужны для разных проектов.
- Рекомендуется в систему контроля версий добавить файл **requirements.txt**, но исключить каталог виртуального окружения (**myenv/**), чтобы избежать загрузки большого количества файлов в репозиторий.
- Следите за обновлением пакетов в виртуальных средах, чтобы воспользоваться последними исправлениями и улучшениями.
- Более совершенные инструменты, такие как pipenv и poetry, предоставляют дополнительные возможности для управления зависимостями. Эти инструменты обеспечивают автоматическое управление виртуальным окружением для ваших проектов и более сложную обработку зависимостей, предоставляя файлы блокировки для обеспечения воспроизводимости среды (см. далее *Примечание переводчика*). Эти инструменты особенно полезны в производственных сценариях, где очень важна согласованность между различными системами и развертываниями.

Примечание переводчика

Детерминированная компиляция (также известная как воспроизводимая сборка) — процесс компиляции программного обеспечения, воспроизведение которого гарантируется. При этом полученный после компиляции двоичный код в точности соответствует исходному коду. Исходный код, скомпилированный с использованием детерминированной компиляции, всегда будет выводить один и тот же двоичный файл вплоть до бита.

Детерминированная сборка — это процесс сборки одного и того же исходного кода с одной и той же средой и инструкциями сборки, когда создаются одни и те же двоичные файлы в любом случае, даже если они сделаны на разных машинах, сохранены в разных каталогах и с разными именами.

Основные принципы работы с интерфейсом командной строки (CLI)

Интерфейс командной строки (CLI) — незаменимый инструмент для разработчиков, особенно при организации и управлении средой разработки внутренних веб-приложений. Этот инструмент обеспечивает эффективное, быстрое и гибкое взаимодействие с компьютером. В инструменте используются текстовые команды, а не графический интерфейс. Знание и владение интерфейсом командной строки может значительно повысить производительность и эффективность работы, и особенно полезно при работе в таких средах, как Linux, где многие задачи разработки могут быть более эффективно решены именно с помощью командной строки.

Описание интерфейса командной строки

Работа с интерфейсом командной строки подразумевает ручной ввод команд в терминал или командную строку для выполнения определенных задач, таких как навигация по файловой системе, управление файлами и каталогами, установка программного обеспечения или управление системными процессами. Эти команды обрабатываются программой-оболочкой, которая интерпретирует команды и выступает в качестве посредника между пользователем и операционной системой.

Основные команды интерфейса командной строки

Ниже перечислены основные команды навигации и управления файловой системой в системе на основе Linux, которая и станет вашей средой разработки.

- `pwd` (Print Working Directory или Показать рабочий каталог).

  Эта команда отображает текущий каталог, в котором вы находитесь, помогая определить ваше нынешнее местоположение в файловой системе.

- **ls** (List или Список).

  Эта команда предназначена для просмотра списка всех файлов и каталогов в текущем каталоге.

  `ls -l` предоставляет подробный список, включая права доступа к файлам, количество ссылок, владельца, группу, размер и дату последнего изменения.

  `ls -a` перечисляет все записи, включая те, что начинаются с точки, но по умолчанию скрыты.

- **cd** (Change Directory или Изменить каталог).

  Эта команда позволяет выполнить смену текущего каталога.

  `cd /path/to/directory` (cd /путь/в/каталог) перемещает вас в указанный каталог.

  `cd ...` перемещает вас на один уровень вверх по каталогу.

  `cd` или `cd ~` (тильда) возвращает вас в ваш домашний каталог.

- **mkdir** (Make Directory или Создать каталог).

  Создание нового каталога.

  `mkdir new_directory` создает новый каталог с именем *new_directory* в текущем каталоге.

- **rmdir** (Remove Directory или Удалить каталог):

  Удаление пустого каталога.

  `rmdir old_directory` удаляет каталог с именем *old_directory*, если он пуст.

- **rm** (Remove или Удалить):

  Удаляет файлы или каталоги.

  `rm file.txt` удаляет файл `file.txt`.

  `rm -r directory` рекурсивно удаляет каталог с именем *directory* и все его содержимое.

> ***Примечание переводчика***
>
> *Рекурсия* (recursion) — это поведение функции, при котором она вызывает сама себя. Такие функции называются рекурсивными. В отличие от цикла, они не просто повторяются несколько раз, а работают «внутри» друг друга.

- **touch** (Create File или Создать файл).

  Создает новый пустой файл или обновляет временную метку существующего файла.

  `touch filename` создает новый файл с именем *filename* или обновляет время его последнего изменения, если такой файл существует.

- **nano**, **vim**, **emacs** (текстовые редакторы):

  С помощью командной строки можно открывать файлы в текстовом редакторе.

  `nano file.txt`, `vim file.txt` и `emacs file.txt` открывают файл `file.txt` в редакторах Nano, Vim и Emacs соответственно.

- **cat** (Concatenate или Конкатенация).

  Выводит содержимое файлов на терминал.

  `cat file.txt` выводит на экран содержимое файла `file.txt`.

  > ***Примечание переводчика***
  > *Конкатенация* — операция по склеиванию объектов линейной структуры, обычно строк. Например, конкатенация слов «микро» и «мир» даст слово «микромир».

- **cp** (Copy или Копия).

  Копирует файлы или каталоги.

  `cp source_file destination_file` копирует *source_file* (исходный файл) в `destination_file` (файл с новым именем).

  `cp -r source_directory destination_directory` рекурсивно копирует *source_directory* (исходный каталог) в *destination_directory* (целевой_каталог).

- **mv** (Move или Переместить).

  Перемещает или переименовывает файлы или каталоги.

  `mv old_name new_name` переименовывает файл или каталог из *old_name* (старое имя) в `new_name` (новое имя).

- **grep** (Global Regular Expression Print или Глобальная печать регулярных выражений).

  Поиск шаблонов в файлах с помощью регулярных выражений.

`Grep 'pattern' file.txt` ищет шаблон *'pattern'* в файле `file.txt` и печатает только строки, в которых этот шаблон найден.

- **chmod** (Change Mode или Режим модификации).

  С помощью этой команды изменяются права доступа к файлу.

  `chmod 755 file.txt` устанавливает разрешения на чтение, запись и выполнение для владельца файла `file.txt` и на чтение и выполнение для группы и других.

- **chown** (Change Owner или Смена владельца).

  Смена владельца и/или группы владельцев файла или каталога.

  `chown user:group file.txt` меняет для файла `file.txt` прежнего владельца на нового вдадельца *user*, а прежнюю группу на новую группу *group*.

Советы по использованию интерфейса командной строки

- Используйте для автоматического заполнения имен файлов и каталогов клавишу <Tab>. Это поможет сэкономить время и уменьшить количество ошибок при вводе.
- Используйте клавиши со стрелками вверх и вниз для просмотра ранее введенных команд. Это позволяет легко повторять или изменять ранее введенные команды.
- Используйте конвейеры (|) и перенаправление (>, >>), чтобы направить вывод одной команды в другую или в файл, комбинируя команды для эффективного выполнения сложных задач.

В среде Linux вы можете значительно упростить процесс разработки внутренних компонентов, используя перечисленные основные команды интерфейса командной строки. Они позволяют автоматизировать процессы, управлять системой и быстро выполнять задания.

Введение в управление версиями с помощью Git

Системы контроля версий — важнейший компонент процесса разработки программного обеспечения, позволяющий разработчикам управлять изменениями, вносимыми с течением времени в исходный код. Этот инструмент незаменим при совместной работе, поскольку позволяет нескольким людям бесконфликтно работать с одной и той же кодовой базой, а также сохраняет историю изменений, которая может быть полезна при отладке и анализе развития проекта. Среди различных систем контроля версий Git является одним из самых популярных и широко используемых инструментов благодаря своей эффективности, гибкости и мощным возможностям создания новых ответвлений.

Знакомство с управлением версиями и Git

По своей сути контроль версий — это метод отслеживания и управления изменениями в программном коде. Системы контроля версий позволяют одновременно работать над одним проектом нескольким разработчикам. Каждый разработчик работает над своей копией кода, а его изменения вносятся в общий репозиторий. Таким образом, обеспечивается полная фиксация того, кто и когда внес изменения, что позволяет отследить конкретные изменения или даже откатиться назад, если это необходимо.

Разработанный Линусом Торвальдсом в 2005 году, Git представляет собой распределенную систему контроля версий. В отличие от централизованных систем контроля версий, созданный на каждом компьютере каталог Git представляет собой полноценный репозиторий с полной историей и возможностью отслеживания версий, не зависящий от доступа к сети или центральному серверу. Git особенно славится своей надежностью, скоростью и эффективностью при работе как с небольшими, так и с очень крупными проектами.

Основные операции Git для разработки внутренних компонентов

Для начала работы с Git изучим несколько основных команд и понятий, которые необходимы для выполнения повседневных задач разработки.

Установка Git

Для установки Git в операционной системе Linux следует выполнить команду:

```
sudo apt-get install git
```

Для проверки результатов установки наберите команду:

```
git --version
```

Конфигурирование Git

Введите свое имя пользователя (*Your Name*) и адрес электронной почты (*your.email@gitforgits.com*) с помощью следующих команд:

```
--global user.name "Your Name"
git config --global user.email "your.email@gitforgits.com"
```

Git внедряет эту информацию в каждую вашу фиксацию.

> ***Примечание редактора***
>
> *Фиксация*, *коммит* (*англ.* commit) — это команда в системе контроля версий Git, которая фиксирует изменения в репозитории. Каждое такое действие создает новую версию файлов, которую можно просматривать, возвращать или изменять в будущем.

Инициализация репозитория

С помощью команд командной строки перейдите в каталог вашего проекта и введите команду `git init`. В результате выполнения этой команды будет создан новый подкаталог **.git**, в котором будут храниться все необходимые файлы репозитория — его скелет. Также инициализируется новый корневой комментарий.

Клонирование репозитория

Чтобы скопировать существующий Git-репозиторий, размещенный в каком-либо месте, например на GitHub, используйте команду:

```
git clone https://github.com/username/repository.git
```

Эта команда создает каталог пользователя *username* со всеми файлами и полной историей ревизий.

Добавление и фиксация файлов

Используйте команду

```
git add <filename>
```

для внесения изменений в конкретный файл *filename*.

Для внесения всех изменений в каталог введите команду:

```
git add
```

Выполните поэтапную фиксацию файлов с помощью команды

```
git commit -m "Commit message"
```

В сообщении *Commit message* описываются все внесенные изменения.

Ветвление и слияние

Ветви используются для разработки изолированных друг от друга функций. Мастер-ветвь — это ветвь, которая создается «по умолчанию» при создании репозитория.

1. Создайте новую ветвь *new-branchname* с помощью команды:

   ```
   git branch new-branchname
   ```

2. Перейдите в эту ветвь с помощью команды:

   ```
   git checkout newbranch-name
   ```

3. По окончании работы над проектом, сохраненным в этой ветви, с помощью команды `git checkout` *master* верните ее в основную ветвь (*master*), а затем выполните команду слияния:

   ```
   git merge new-branch-name
   ```

Размещение изменений

Чтобы отправить свои комментарии в удаленный репозиторий, используйте команду

```
git push origin master
```

где *origin* — это имя вашего удаленного репозитория, присвоенное по умолчанию, а *master* — это имя вашей ветви.

Извлечение обновлений

Чтобы получить изменения из удаленного репозитория и объединить их на вашей локальной машине, используйте команду

```
git pull
```

Обработка конфликтов при слиянии

Конфликты возникают при слиянии ветвей с конкурентными изменениями. Git предложит вам разрешить конфликты вручную, отредактировав файлы и выполнив команду

```
git add <file>
```

для указания способа разрешения конфликта.

Использование журнала Git

Команда `git log` выводит хронологическую историю зафиксированных изменений для текущей ветки.

Для краткого просмотра истории используйте команду

```
git log --oneline
```

Внедрив Git в рабочий процесс разработки, вы сможете более эффективно управлять работой над проектом. Для каждого проекта по разработке внутренних компонентов Git — это не просто средство защиты, это мощный инструмент для совместной работы и управления изменениями.

Библиотека Python Refresher: синтаксис, структуры данных и функции

Синтаксис Python

Python отличается своей читабельностью и простотой, что делает его наилучшим выбором как для начинающих, так и для опытных разработчиков. Синтаксис Python интуитивно понятен и приближен к человеческому языку, и в этом его привлекательность. Далее перечислены некоторые ключевые особенности.

Отступы

Для создания отступов в Python используется клавиша <Пробел>. С помощью отступов определяется область действия. Например, области действия циклов, функций и классов. В других языках программирования для этой цели часто используются фигурные скобки. Это требование к отступам помогает сделать код Python хорошо читаемым.

```
if 10 > 5:
print("10 is greater than 5")
```

> ***Примечание переводчика***
>
> Для первого отступа в Python лучше использовать 4 пробела. А далее добавлять их группами по 4 штуки. Эти простые правила помогут избежать ошибки Unexpected Indent Python (Неожиданный отступ Python).

Переменные

В языке Python переменные создаются при присвоении им значения. Python является динамически типизированным языком программирования, что означает, что вам не нужно объявлять тип переменной при ее объявлении.

```
x = 5
y = "Hello, World!"
```

> ***Примечание переводчика***
>
> Переменная — это простейшая именованная структура данных, в которой может быть сохранен промежуточный или конечный ре-

зультат работы программы. Переменные в Python следует именовать в нижнем регистре, а слова разделять нижним подчеркивание. Например: `max_age`, `init_state`. После объявления переменной мы можем ее использовать в любом выражении.

Комментарии

В Python для обозначения комментария в коде используется символ решетки #.

```
# This is a comment (Это комментарий)
print("Hello, World!")
```

Структуры данных

Python предлагает несколько встроенных структур данных, которые характеризуются надежностью и широкими возможностями. Далее перечислены наиболее часто используемые структуры данных.

Списки

Списки — это упорядоченные изменяемые коллекции, допускающие дублирование элементов.

```
fruits = ["apple", "banana", "cherry"]
print(fruits[1]) # Вывод: banana
fruits.append("orange") # Добавляет "orange" в конец списка
```

Примечание переводчика

Список в Python — это упорядоченный изменяемый набор объектов произвольных типов, нумерация которых начинается с 0.

Кортежи

Кортежи — это упорядоченные и неизменяемые коллекции, допускающие дублирование членов. Кортежи обозначаются круглыми скобками.

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1]) # Вывод: banana
```

Примечание переводчика

В кортеже нельзя заменить значение элемента, добавить или удалить элемент. Простыми словами, кортеж — неизменяемый список.

Множества

Множества — это неупорядоченные, неиндексированные библиотеки без дубликатов. Отлично подходят для тестирования элементов и устранения дубликатов.

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset) # Вывод: True
```

Словари

Словари — неупорядоченные, изменяемые, индексируемые коллекции, в которых нет повторяющихся элементов. Каждый элемент словаря состоит из пары *ключ* : *значение*.

```
thisdict = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
print(thisdict["model"]) # Вывод: Mustang
```

Функции

Функции — это ключевой инструмент для разделения кода Python на модули. При использовании функций код становится более читабельным и пригодным для повторного использования. Функции определяются с помощью ключевого слова `def`.

Определение функции

Ниже описано, как определить функцию Python:

```
def greet(name):
return "Hello, " + name
```

Примечание переводчика

Для определения функции в Python используется ключевое слово `def`, за которым следует имя функции и круглые скобки. В скобках указываются аргументы функции (если они есть), а после двоеточия начинается блок кода функции. Блок кода функции обычно должен быть с отступом.

Вызов функции

Для вызова функции следует использовать имя функции, заключенное в круглые скобки.

```
message = greet("Alice")
print(message) # Вывод: Hello, Alice
```

Параметры

Функции принимают параметры, которые указываются после имени функции. Параметры заключаются в круглые скобки. Вы также можете задать значения параметров по умолчанию.

```
def greet(name, greeting="Hello"):
return greeting + ", " + name
```

Примечание переводчика

Параметр — это переменная, с помощью которой присваивается входящее в функцию значение.

Аргументы ключевых слов

Чтобы гарантировать, что каждый аргумент будет сопоставлен с соответствующим параметром, даже если они расположены не по порядку, при вызове функций следует использовать ключевое слово arguments.

```
print(greet(name="Bob", greeting="Hi")) # Вывод: Hi, Bob
```

Примечание переводчика

Аргумент — это значение, которое передается в функцию при ее вызове. Аргументом может быть любой объект. Аргумент может передаваться как в литеральной форме, так и в виде переменной. Значения в позиционных аргументах подставляются согласно позиции имен аргументов. Функции в Python могут принимать имено-

ванные аргументы. В этом случае соответствие между значениями и именами аргументов определяется по именам, а не по позициям аргументов.

Произвольные аргументы

Если вы не знаете, какое количество аргументов будет передано вашей функции, добавьте в определение функции перед именем параметра символ звездочки `*`.

```
def make_smoothie(*fruits):
for fruit in fruits:
print("Adding:", fruit)
make_smoothie("apple", "banana", cherry")
```

Лямбда-функции

Лямбда-функции — это небольшие анонимные функции, определяются с помощью ключевого слова `lambda`. Лямбда-функции могут иметь любое количество аргументов, но только одно выражение.

```
double = lambda x: x * 2
print(double(5)) # Вывод: 10
```

Примечание переводчика

Лямбда-функции в Python часто используются там, где нужно использовать небольшую, одноразовую функцию. Синтаксис для определения лямбда-функции в Python следующий:

```
Lambda arguments : expression
```

Здесь *arguments* — это список входных аргументов функции, который разделен запятыми. А *expression* — это тело функции, значение которой возвращает лямбда-функция.

Обработка ошибок

Обработка ошибок в Python выполняется с помощью исключений. Блок кода, который может привести к ошибке, следует поместить внутрь блока `try`. Блок `except` содержит код, который обрабатывает ошибку.

```
try:
print(x)
except NameError:
print("Variable x is not defined")
except:
print("Something else went wrong")
```

Модули и пакеты

Модули в Python — это обычные файлы Python с расширением **.py**, в которых содержится код Python, пригодный для импорта в другие файлы Python. Чтобы сгруппировать несколько модулей вместе, Python использует пакеты, представляющие собой каталоги, содержащие специальный файл **__init__.py**.

Импорт модулей

Импорт используется для добавления любого файла в Python в качестве модуля.

```
import math
print(math.sqrt(16)) # Output: 4.0
```

Импорт с помощью псевдонимов

При импорте модуля можно создать псевдоним, используя ключевое слово `as`.

```
import datetime as dt
today = dt.date.today()
```

Оператор импорта *from*

Если вам необходимо получить только определенные элементы из модуля, вы можете использовать оператор `from`, чтобы импортировать эти элементы непосредственно в свое пространство имен.

```
from math import sqrt
print(sqrt(16)) # Output: 4.0
```

Знание основных принципов языка Python позволит вам создавать более эффективные внутренние компоненты приложений.

Разработчикам необходимо хорошо разбираться в таких особенностях Python, как управление данными с помощью встроенных структур, построение функций для инкапсуляции функциональности или обработка ошибок и исключений.

Передовой опыт программирования на Python

Использование передового опыта программирования на языке Python имеет решающее значение для написания эффективного, читабельного и поддерживаемого кода, особенно при разработке внутренних компонентов, где часто возникают проблемы сложности и масштабируемости. Далее приведены несколько примеров из передового опыта программирования на Python.

Поддержка принципов «Дзен Python»

«Дзен Python» — это набор тезисов, отражающих философию Python. С этими тезисами вы можете ознакомиться, набрав в вашем интерпретаторе Python команду `import this`. Здесь подчеркивается простота, читабельность и понятие «один очевидный способ сделать так», что помогает сделать код интуитивно понятным и легким в сопровождении.

Соблюдение стандарта PEP 8

PEP 8 — это руководство по оформлению кода Python. В нем содержатся рекомендации по форматированию и написанию кода Python, включающие такие моменты, как соглашения об именовании, отступы, длина строки, пробельные символы, организация импорта и комментарии. Соблюдение PEP 8 улучшает читаемость и согласованность кода, что очень важно в среде совместной работы.

Отступы: используйте 4 пробела для каждого уровня отступа.

Длина строки: ограничьте количество строк до 79 символов.

Соглашения об именовании: используйте `CamelCase` для классов, `snake_case` для функций и переменных, `UPPER_CASE` для констант.

Написание документальных строк

Документальные строки используются для документирования классов, методов, функций и модулей Python. Эти строки заключаются в тройные кавычки и располагаются сразу после определения функции, метода или класса.

```
def greet(name):
"""
Greet a person with their name.
Parameters:
name (str): The name of the person
to greet.
Returns:
str: A greeting message.
"""
return f"Hello, {name}!"
```

```
def greet(имя):
"""
Приветствует человека по имени.
Параметры:
имя (str): Имя человека, которого
нужно поприветствовать.
Возвращает:
str: Приветствие.
"""
return f «Привет, {имя}!»
```

Использование документальных строк помогает другим разработчикам понять, что делает код, не погружаясь в детали, а это способствует лучшему сотрудничеству и сопровождению.

Использование встроенных функций и библиотек Python

Python содержит богатую стандартную библиотеку и множество встроенных функций, которые помогут вам выполнять привычные задачи более эффективно. С помощью этих функций вы будете создавать более качественный, чистый и эффективный код за короткое время.

- Функции типа `map()`, `filter()`, `sum()`, `min()`, `max()` могут заменить циклы, создаваемые вручную, и обычно являются более эффективными.
- Прежде чем писать собственную версию функции, проверьте, нет ли такой функции в библиотеке. Например, для разбора JSON можно использовать функцию `json`, для работы с датами — функцию `datetime`, а для системных операций — функции `os` и `sys`.

Использование списков и генераторов выражений

Списочные выражения (list comprehension) и генераторные выражения (generator expression) обеспечивают краткий и понятный способ создания списков или итераторов. Данный способ гораздо проще и удобнее, чем циклическое создание списков.

```
# Списочное выражение
squares = [x**2 for x in range(10)]
# Генераторное выражение
sum_of_squares = sum(x**2 for x in range(10))
```

> ***Примечание переводчика***
>
> *Генераторные выражения* — это упрощенный вариант функций-генераторов, также создающих генераторы. Функция-генератор отличается от обычной функции тем, что вместо команды `return` в ней используется инструкция `yield`. И если `return` завершает работу функции, то инструкция `yield` лишь приостанавливает ее, при этом возвращается какое-то значение.
>
> *Итератор* — это интерфейс, упрощающий навигацию по элементам объекта, как правило, представляющего собой некоторую коллекцию (список, словарь и т. п.). Итератор представляет собой объект-перечислитель, который для данного объекта поочередно выдает следующий элемент или генерирует исключение, если элементов больше нет. Основное место применения итераторов — это цикл `for`. При каждой итерации цикла происходит обращение к итератору, содержащемуся в строке или списке, с требованием выдать следующий элемент.

Обработка ошибок с помощью исключений

Корректная обработка ошибок очень важна для создания надежных приложений. В Python для обработки ошибок применяют исключения, грамотное использование которых может предотвратить аварийное завершение работы приложения и предоставить пользователю или разработчику содержательные сообщения об ошибках.

- Используйте конкретные исключения, а не общее условие `except`.
- Для отлова потенциальных ошибок используйте блоки `try-except`.

- Всегда очищайте ресурсы с помощью `finally` или менеджера контекста.

> ***Примечание переводчика***
>
> Блок `finally` в Python используется для выполнения кода, который должен выполниться в любом случае — независимо от того, было ли выброшено исключение в блоке `try` или нет.

```
try:
process_file(file)
except FileNotFoundError:
print("File does not exist")
finally:
file.close()
```

Использование менеджеров контекста для управления ресурсами

Менеджеры контекста Python играют важную роль в управлении ресурсами, такими как файловые потоки, которые после завершения их использования необходимо правильно закрывать. Применение оператора `with` обеспечивает своевременную очистку ресурсов.

```
with open('example.txt', 'r') as file:
contents = file.read()
```

Функция должна выполнять одну задачу и иметь минимально возможный размер

У каждой функции должна быть только одна, четко определенная задача. Таким образом, функции можно использовать повторно, тестировать и отлаживать. Если одна функция выполняет много задач, подумайте о том, чтобы разделить эту функцию на несколько более маленьких функций.

Избежание преждевременной оптимизации

Несмотря на то что код должен эффективно выполнять свои задачи, избегайте преждевременной оптимизации. Сначала создай-

те понятный и корректный код. Затем используйте инструменты профилирования, чтобы найти узкие места и оптимизировать именно эти части кода.

Использование контроля версий

Использование системы контроля версий, такой как Git, помогает не только контролировать изменения в кодовой базе, но и документировать причины изменений кода с помощью сообщений о фиксации. Это очень важно для совместной работы и ведения истории кода.

Тестирование своего кода

Проведение тестов необходимо, чтобы убедиться, что ваш код ведет себя так, как ожидается. Используйте для Python модуль тестирования `unittest`. Или примените для составления тестов, которые проверяют каждую часть вашего приложения, сторонние библиотеки типа `pytest`.

```
import unittest
class TestSum(unittest.TestCase):
def test_sum(self):
self.assertEqual(sum([1, 2, 3]), 6, "Should be 6")
if __name__ == '__main__':
unittest.main()
```

Разработчики Python, если воспользуются приведенными ранее советами, могут обеспечить не только функциональность кода, но и его чистоту, удобство сопровождения и простоту восприятия. Поскольку с добавлением новых функций сложность системы может резко возрасти, этот передовой опыт очень важен при разработке внутренних компонентов.

Резюме

В этой главе мы рассмотрели основы создания внутренних компонентов на языке программирования Python. Для начала было дано определение процесса разработки внутренних компонентов

и объяснено, почему Python благодаря своей простоте и отказоустойчивости является наилучшим выбором для выполнения этих задач. Далее мы затронули все вопросы, связанные с настройкой среды разработки, включая установку Visual Studio Code и Python и настройку с помощью этих инструментов операционной системы Linux для создания рабочего пространства. Мы также рассмотрели вопросы создания и обслуживания виртуальной среды для бесконфликтной работы с зависимостями конкретного проекта.

Также нами была изучена работа с интерфейсом командной строки (CLI). В этом разделе мы показали основные команды для управления файловыми системами и запуска программ. Для разработки внутренних компонентов это очень важно. Затем мы познакомились с Git, системой контроля версий, и узнали о преимуществах этой системы для управления версиями проектов и совместной работы в команде. Создание репозитория, внесение изменений, работа с ветвями и слияние кода — все это необходимые навыки для поддержания организованной кодовой базы.

Наконец, чтобы убедиться, что у вас есть прочный фундамент в программировании, мы еще раз рассмотрели синтаксис, структуры данных и функции Python. В этой главе также были приведены примеры передового опыта работы с Python, особенно в части написания эффективного, хорошо структурированного и простого в сопровождении кода. Использование содержательной документации, правильная обработка ошибок и исключений, а также использование встроенных функций Python — все это относится к самому передовому опыту.

В данной главе мы рассмотрели все, что необходимо знать для создания среды разработки внутренних компонентов на базе Python: от инструментов и методологий до передового опыта кодирования и примеров хорошо построенных внутренних приложений. В последующих главах будут рассмотрены более сложные темы, а этот фундаментальный материал заложит основу для них.

ГЛАВА 2
Создание первого веб-приложения с помощью Flask

Введение

Из этой главы вы узнаете, как создать веб-проект с помощью Flask, надежного и легкого веб-фреймворка, предназначенного для создания веб-приложений на языке программирования Python. Цель этой главы — дать подробный обзор функций и компонентов, необходимых для создания веб-приложения с нуля. Сначала мы расскажем вам о фреймворке Flask, рассмотрим его главные особенности. Далее вы узнаете, что по причине его простоты и универсальности этот фреймворк стал одним из лучших инструментов для быстрой разработки веб-приложений. Следующий шаг — подготовка среды Flask, в ходе которой будут установлены все необходимые библиотеки и инструменты, что позволит начать разработку приложения.

В процессе обучения вы освоите основы маршрутизации и представлений в веб-приложениях. В том числе научитесь работать с различными URL-адресами и связывать их с функциями Python. Это позволит вам формировать динамический контент на основе запросов пользователей. Следующий шаг — знакомство со статическими файлами и шаблонами. В этом разделе вы узнаете, как создавать и управлять HTML-шаблонами, совместимыми с Python и отображающими информацию в удобном для восприятия виде.

Для работы интерактивных приложений необходимо иметь возможность обработки загрузки файлов и отправки форм. Книга научит вас принимать пользовательский ввод, обрабатывать и правильно реагировать на него. Также вы узнаете, как защитить и проверить данные, чтобы ваше приложение могло противостоять

вредоносному вводу. Еще один важный момент, на который мы обратим внимание, — интеграция баз данных. Расширение для Flask, Flask-SQL-Alchemy предоставляет простые, но мощные инструменты для взаимодействия с базами данных, позволяя эффективно хранить, извлекать, изменять и удалять данные.

Вы познакомитесь с несколькими наиболее популярными расширениями, которые позволяют расширить возможности Flask, не загромождая кодовую базу, и узнаете, как использовать эти расширения для наращивания функциональности ваших приложений, созданных на основе Flask. И наконец, что не менее важно, вы узнаете о различных тактиках развертывания и платформах, на которых можно развернуть ваше приложение, что позволит сделать его доступным для людей по всему миру.

Из этой главы вы почерпнете все, что вам нужно знать о фреймворке Flask, чтобы создавать, поддерживать и запускать полнофункциональные веб-приложения. Эти знания заложат основу для использования в будущем более сложных функций Flask и создания более крупных приложений.

Основы Flask

Что представляет собой Flask?

Разработанный Армином Ронахером и выпущенный в 2010 году, фреймворк Flask основан на инструментах Werkzeug и WSGI и шаблонизаторе Jinja2. Flask — это легкий и гибкий веб-фреймворк для создания веб-приложений на языке Python, который особенно хорошо подходит для написания небольших и средних по размеру приложений, а также для разработчиков, которые хотят получить детальный контроль над своими компонентами.

> ***Примечание переводчика***
>
> Werkzeug — это набор библиотек, которые можно использовать для создания WSGI-совместимых веб-приложений на Python.
>
> WSGI (Web Server Gateway Interface или интерфейс шлюза веб-сервера) — это интерфейс между веб-сервером и веб-приложением, разработанный на базе Python. Он необходим для рабо-

ты веб-приложений, написанных на языке Python, потому что веб-сервер не может напрямую взаимодействовать с Python.

Шаблонизатор — это инструмент, который упрощает создание разметки, позволяет разделить эту разметку на компоненты и связать с данными. Главное преимущество шаблонизаторов — они избавляют от необходимости писать повторяющийся код несколько раз. Шаблонизаторы помогают решать следующие задачи:

- создание базовой разметки страниц;
- внесение правок одновременно на нескольких страницах или компонентах;
- изменение контента в блоках;
- добавление, удаление или перенос блоков на страницах сайта;
- создание повторяющегося контента, например карточки товара в каталоге.

Flask содержит минимум встроенных функций, что делает его легким, простым и расширяемым, и поэтому он хорошо подходит для разработчиков, которые предпочитают создавать свои приложения с нуля. Придерживаясь принципов простоты и детального контроля, фреймворк предлагает инструменты, которые необходимы для запуска веб-приложения с минимумом настроек, но при этом он позволяет использовать расширения для добавления дополнительной функциональности, например объектно-реляционного отображения, проверки форм и обработки загрузок.

Основные возможности Flask

Простота

Flask рассматривается как фреймворк, наиболее подходящий для программирования на языке Python. И от веб-фреймворка Django отличается минимализмом и легкостью в освоении. С помощью Flask вы можете создать простое веб-приложение, состоящее всего из нескольких строк кода.

Гибкость

Flask предоставляет возможность самостоятельно выбирать, как реализовать то или иное решение. Фреймворк позволяет исполь-

зовать предпочитаемые вами инструменты для решения различных задач, таких как взаимодействие с базой данных, маршрутизация URL и отрисовка шаблонов.

Сервер разработки и отладчик

В состав фреймворка Flask входит встроенный сервер разработки и отладчик. Сервер позволяет тестировать приложения локально и автоматически перезагружать их при обнаружении изменений в коде. Отладчик выдает полезные сообщения об ошибках и трассировку стека, если что-то идет не так.

> ***Примечание переводчика***
>
> *Трассировка стека* (стек-трейс) — это последовательность вызовов методов, которая привела к возникновению исключения, т. е. ошибки. Эта информация очень полезна при отладке кода, т. к. позволяет точно определить, в каком месте кода произошла ошибка.

Основан на Юникоде

Flask без дополнительных настроек поддерживает Юникод сразу после установки фреймворка, что позволяет использовать в приложениях символы, отличные от ASCII, а это очень важно для работы с международными приложениями.

Документация

Flask хорошо документирован, располагает множеством доступных ресурсов и поддерживается многочисленным сообществом. Официальная документация тщательно проработана и является хорошей отправной точкой как для новичков, так и для опытных разработчиков.

Запуск и работа с Flask

Чтобы начать работу с Flask, вам понадобится установленный на вашем компьютере язык Python. Flask совместим с Python версии 3.6 и более новыми. Далее описано, как настроить базовое приложение Flask.

1. Установите Flask с помощью пакетного менеджера Python pip:

```
pip install Flask
```

2. После установки Flask вы можете создать простое приложение "Hello, World!":

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
return 'Hello, World!'
if __name__ == '__main__':
app.run(debug=True)
```

Ниже показано, что делает каждая часть кода.

- `from flask import Flask` импортирует класс Flask.
- `app = Flask(__name__)` создает экземпляр класса Flask.
- `@app.route('/')` — это декоратор, который указывает Flask, какой URL должен вызывать функцию.

> ***Примечание переводчика***
>
> *Декоратор* в Python — это инструмент, который позволяет модифицировать поведение функций или классов без изменения их кода. Декораторы представляют собой функции высшего порядка, т. е. функции, которые принимают другие функции в качестве аргументов и возвращают новые функции.

- `def hello_world():` определяет функцию, которая вызывается по URL.
- `return 'Hello, World!'` или возвращаемый текст "Hello, World!" — это ответ с URL-адреса.
- `app.run(debug=True)` запускает приложение на локальном сервере разработки, а аргумент `debug=True` позволяет отображать возможные ошибки Python на веб-странице.

> ***Примечание переводчика***
>
> Python поставляется со встроенным отладчиком под названием PDB (Python Debugger). Он позволяет вам приостанавливать выполнение вашего кода на Python, проверять переменные и пошагово просматривать ваш код построчно, чтобы находить и устра-

нять проблемы. В то время как инструкции печати и ведение журнала полезны для базовой отладки, PDB выводит отладку на новый уровень, позволяя вам вмешиваться и анализировать ваш код в режиме реального времени.

Маршрутизация

Маршрутизация в Flask осуществляется с помощью декоратора `app.route`. Этот декоратор привязывает функцию к адресу URL. Ниже приведен пример маршрутизации:

```
@app.route('/greet')
def greet():
return 'Hello from Flask!'
```

Вы также можете сделать части URL-адреса динамическими и к одной функции прикрепить несколько правил:

```
@app.route('/post/<int:post_id>')
def show_post(post_id):
return 'Post %d' % post_id
```

В данном случае `<int:post_id>` указывает, что `post_id` должен быть целым числом.

Шаблоны

В качестве шаблонизатора Flask использует библиотеку Jinja2. Шаблон — это файл, который содержит статические данные, а также заполнители для динамических данных. В маршруте Flask шаблон может быть отрисован и возвращен клиенту.

Примечание переводчика

Jinja2 — это Python-библиотека для рендеринга шаблонов. Она является стандартом при написании веб-приложений на Flask и популярной альтернативой встроенной системе шаблонов Django. Jinja2 позиционирует себя как инструмент для дизайнеров и верстальщиков, упрощающий верстку и отделяющий ее от разработки. Библиотека Jinja2 расширяема, и многие возможности (например, интернационализация и управление циклами) реализованы именно как расширения.

Заполнитель — это текстовые подсказки или краткие описания, которые помогают пользователю при заполнении формы, напри-

мер для прохождения регистрации на сайте или оплаты товара в интернет-магазине. Заполнитель находится внутри поля для какого-либо блока: ввода адреса электронной почты, ввода пароля, ввода телефона и т. д. Текст-заполнитель сообщает, какие именно данные относятся к каждому конкретному полю формы, и дает дополнительные подсказки, описания или примеры данных, которые необходимо вводить. Как правило, он автоматически пропадает сразу же после того, как пользователь вводит первый символ в поле.

Ниже приведен пример рендеринга шаблонов:

1. Сначала создайте каталог с именем **templates** в основной директории приложения.
2. Внутри этого каталога создайте HTML-файл с именем **index.html**.
3. Добавьте в **index.html** следующее содержимое:

```
<h1>Hello {{ name }}!</h1>
```

4. Отрисовка шаблона:

```
from flask import render_template
@app.route('/hello/<name>')
def hello(name=None):
return render_template('index.html', name=name)
```

Функция `render_template` принимает имя файла шаблона и изменяемый список аргументов шаблона и возвращает тот же самый шаблон, но с заменой всех заполнителей в шаблоне на фактические значения.

Приложения Flask легко отлаживать с помощью встроенного сервера разработки. Когда вы с помощью команды `app.run(debug=True)` запускаете приложение, Flask, если ваше приложение обнаружит ошибки, запустит в браузере интерактивный отладчик. Flask предоставляет вам все необходимое для создания API, веб-приложений или даже для изучения разработки внутренних компонентов с минимальными затратами на подготовку и настройку.

Настройка окружения Flask

Для начала разработки веб-приложения вам необходимо создать среду Flask. Процесс настройки включает в себя несколько шагов. Сейчас мы уделим внимание организации проекта, управлению зависимостями и настройке критических сервисов.

Создание проекта

Правильно организованная структура проекта очень важна для поддержания кодовой базы по мере роста приложения. Ниже приведена типичная структура проекта Flask:

- `/app` здесь содержится код вашего приложения.
  - `__init__.py` инициализирует ваше приложение Flask и объединяет остальные компоненты.
  - `views.py` содержит маршруты и представления для вашего приложения.
  - `models.py` содержит модели баз данных.
  - `/static` здесь хранятся статические файлы, такие как JavaScript, CSS и изображения.
  - `/templates` содержит шаблоны Jinja2.
- `/venv` — каталог для виртуальной среды (уже настроенной, как было указано ранее).
  - `requirements.txt` — файл, в котором перечислены все зависимости вашего приложения.
  - `config.py` содержит конфигурационные переменные и настройки.
  - `run.py` — точка входа для запуска вашего приложения.

Настройка Flask

Установка Flask

Для начала убедитесь, что Flask уже установлен в вашей среде Python. Как уже было сказано ранее, вы можете установить Flask с помощью менеджера пакетов Pip:

```
pip install Flask
```

Создание приложения Flask

В файле **__init__.py** настройте ваше приложение Flask:

```
from flask import Flask
app = Flask(__name__)
from app import views
```

Этот скрипт инициализирует объект приложения Flask как `app`. Приложение будет импортировать `views` (представления) из вашего приложения, которые вы определите в файле **views.py**.

Создание простого представления

В файле **views.py** создайте простой маршрут:

```
from app import app
@app.route('/')
def index():
return "Hello Flask!"
```

Управление зависимостями с помощью Pip

Для более эффективной работы с зависимостями проекта вам следует поддерживать файл **requirements.txt**. Каждый раз, когда вы устанавливаете новый пакет, обновляйте этот файл:

```
pip freeze > requirements.txt
```

Этот файл позволяет любому, кто устанавливает проект, легко смонтировать все необходимые зависимости с помощью:

```
pip install -r requirements.txt
```

Настройка приложения Flask

Параметры конфигурации обеспечивают более эффективное управление различными средами (разработка, тестирование, производство).

Создание файла конфигурации

Создайте файл **config.py** для хранения параметров конфигурации. Здесь будут сохранены URL-адреса баз данных, секретные ключи, ключи API и другие параметры окружения:

```
import os
class Config(object):
SECRET_KEY = os.environ.get('SECRET_KEY') or 'you-will-never-guess'
DEBUG = False
TESTING = False
class ProductionConfig(Config):
DATABASE_URI = 'mysql://user@localhost/foo'
class DevelopmentConfig(Config):
DEBUG = True
DATABASE_URI = 'sqlite:///development.db'
class TestingConfig(Config):
TESTING = True
DATABASE_URI = 'sqlite:///testing.db'
```

Загрузка конфигурации

В файле **__init__.py** настройте приложение на использование этих параметров в зависимости от текущего окружения:

```
app.config.from_object('config.DevelopmentConfig')
```

Запуск приложения Flask

Чтобы запустить приложение Flask, используйте командную строку Flask или создайте скрипт.

Командная строка Flask

Установите переменные окружения, чтобы указать Flask, как импортировать ваше приложение, а затем запустите его:

```
export FLASK_APP=run.py
export FLASK_ENV=development
flask run
```

Использование скрипта

Создайте скрипт **run.py** в корне вашего проекта:

```
from app import app
if __name__ == '__main__':
app.run(debug=True)
```

Затем вы можете запустить свое приложение, выполнив команду

```
python run.py
```

Благодаря организованности проекта, тщательному управлению зависимостями и эффективному использованию конфигураций вы сможете обеспечить бесперебойный процесс разработки и готовность вашего приложения к работе в различных средах.

Маршрутизация и представления

Далее мы рассмотрим наиболее важную часть любого веб-приложения: представления и маршрутизацию. Когда пользователь переходит по определенному в Flask URL-адресу, функция маршрутизации определяет, что должно отобразить приложение. Для ответа на эти запросы используются функции, называемые представлениями.

> ***Примечание переводчика***
>
> Представление — это виртуальная логическая таблица, представляющая собой поименованный запрос, который при использовании представления будет представлен как подзапрос. В отличие от обычных таблиц реляционной базы данных, представление не является самостоятельной частью набора данных, хранящихся в базе. Изменение данных в реальной таблице базы данных немедленно отражается в содержимом всех представлений, построенных на основании этой таблицы.
>
> Маршрутизация — это процесс определения пути следования информации в сетях связи. Маршрутизация предназначена для приема пакета от одного устройства и передачи его другому устройству через другие сети.

Виды маршрутизации:

- Прямая маршрутизация — отправитель в определенной IP-сети может напрямую передавать кадры любому получателю в той же сети.
- Косвенная маршрутизация — происходит в том случае, если отправитель и получатель находятся в разных IP-сетях. Отправитель передает пакеты маршрутизатору для доставки их через распределенную сеть.

Определение маршрутов

Маршрут в Flask — это и есть URL-путь к определенной функции в программном коде, написанном на языке Python. Эта функция известна как «представление». Когда Flask получает запрос от клиента (например, веб-браузера), он сопоставляет URL с заранее определенным маршрутом, а затем выполняет связанную с ним функцию представления, которая генерирует ответ. Ниже приведен пример определения базового маршрута:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def home():
return "Welcome to the Homepage!"
```

В этом коде `@app.route('/')` — это декоратор, который указывает, с помощью какого URL Flask должен вызывать следующую за декоратором функцию. В данном случае функция `home` связана с корневым URL сайта.

Динамические маршруты

Нередко приложение должно обрабатывать динамический контент, который меняется в зависимости от URL-адреса. Например, вам может понадобиться отобразить информацию о пользователе или конкретном продукте, идентифицированном по идентификатору в URL. Ниже описано, как можно определить динамические маршруты в Flask:

```
@app.route('/user/<username>')
def show_user_profile(username):
# показать профиль для этого пользователя
return f'User {username}'
@app.route('/post/<int:post_id>')
def show_post(post_id):
# показать пост с заданным id, id - целое число
return f'Post {post_id}'
```

В приведенном выше фрагменте кода `<username>` (имя пользователя) и `<int:post_id>` являются переменными частями URL. Flask при выполнении маршрутизации запросов в этих частях URL будет принимать любые значения и передавать их в качестве аргументов ключевых слов в соответствующие функции представления.

Методы HTTP

По умолчанию на `GET`-запросы отвечают маршруты Flask. Однако, если вы хотите использовать различные HTTP-методы, такие как `POST`, `PUT` или `DELETE`, вы можете указать это в определении маршрута с помощью аргумента `methods`.

```
@app.route('/login', methods=['GET', 'POST'])
def login():
if request.method == 'POST':
return do_the_login()
else:
return show_the_login_form()
```

То есть реакция функции на запросы `GET` и `POST` будет различной. Тип выполняемого HTTP-запроса содержится в `request.method`, и логика работы будет меняться в зависимости от указанного значения.

Создание URL-адресов

Чтобы создать URL-адреса для определенной функции, следует воспользоваться функцией `url_for()`. Эта функция генерирует URL, используя имя и аргументы функции.

```
from flask import url_for
@app.route('/')
def index():
return 'index'
@app.route('/login')
def login():
return 'login'
@app.route('/user/<username>')
def profile(username):
return f'{username}\'s profile'
with app.test_request_context():
print(url_for('index'))
print(url_for('login'))
print(url_for('login', next='/'))
print(url_for('profile', username='John Doe'))
```

Функция `url_for('index')` сгенерирует URL-адрес представления под именем `index`. Это особенно полезно для того, чтобы избежать жесткого кодирования URL в ваших шаблонах и скриптах, что делает ваше приложение более простым в обслуживании.

Обработка ошибок

Корректная обработка ошибок имеет решающее значение для улучшения удобства работы пользователей. Flask позволяет легко определять пользовательские страницы с ошибками для различных типов ошибок.

```
@app.errorhandler(404)
def page_not_found(error):
return "This page does not exist.", 404
@app.errorhandler(500)
def internal_server_error(error):
return "Internal server error.", 500
```

Эти обработчики позволяют легко возвращать пользователю понятные сообщения об ошибках, когда что-то идет не так.

Эффективное сочетание маршрутов и представлений

Преимущество Flask заключается в его простоте и гибкости функций маршрутизации и представлений. Их эффективное сочетание позволяет:

- создавать динамический контент, адаптирующийся к данным или запросам пользователей;
- безопасно и надежно работать с формами и пользовательским вводом;
- создавать RESTful-интерфейсы, которые могут взаимодействовать с другими сервисами или элементами интерфейса.

> ***Примечание переводчика***
>
> RESTful API (Интерфейс передачи репрезентативного состояния) помогает разработчикам создавать, читать, обновлять и удалять ресурсы на сервере, следуя архитектурным ограничениям REST — набору руководящих принципов, ориентированных на крупномасштабные распределенные системы. API-интерфейсы RESTful используют стандартные методы HTTP, такие как GET, POST, PUT и DELETE. Они облегчают взаимодействие клиентов (например, веб-браузеров или мобильных приложений) и серверов. Основная цель RESTful API — обеспечить совместимость различных программных приложений, что значительно упрощает их интеграцию и совместную работу.

По мере разработки более сложных приложений может возникнуть необходимость в улучшении организации маршрутов и представлений. Для этого Flask поддерживает шаблоны, которые позволяют организовать ваше приложение в виде компонентов, которые можно легко использовать повторно или настраивать.

Шаблоны и статические файлы

Теперь, когда мы рассмотрели основы Flask-представлений и маршрутизации, давайте перейдем к управлению шаблонами и статическими файлами. Если вы хотите, чтобы ваше веб-приложение быстро реагировало на запросы и было интересным для

пользователей, а также предоставляло динамический контент, у вас должны быть эти компоненты.

> ***Примечание переводчика***
>
> *Шаблон* — это внешний вид сайта, в котором определяется расположение различных элементов, художественный стиль и способ отображения страниц. Включает в себя программный HTML-код, графические элементы, таблицы стилей, дополнительные файлы для отображения контента. Может также включать в себя шаблоны компонентов, шаблоны готовых страниц и сниппеты. Сниппет — это уменьшенная до небольших размеров копия изображения или фрагмент страницы сайта, который выводится в поисковой выдаче.

Описание шаблонов

Шаблоны в Flask — это эффективная функция, используемая для создания динамического HTML-контента. Flask, работающий на движке шаблонов Jinja2, позволяет создавать HTML-шаблоны с заполнителями для динамического содержимого, которое ваше приложение может генерировать, взаимодействуя с пользователем или другими входными данными.

Шаблоны в Flask обычно представляют собой HTML-файлы с дополнительными заполнителями для динамического содержимого. Эти заполнители задаются с использованием синтаксиса шаблона Jinja2. Например, вы можете передавать переменные из функции просмотра Flask в шаблон и использовать эти переменные для заполнения полей в HTML.

Ниже приведено описание того, как настроить и использовать шаблоны:

1. Создайте в каталоге проекта папку для шаблонов с именем **templates**. Flask автоматически произведет поиск шаблонов в папке с именем templates.
2. Внутри папки **templates** создайте HTML-файл, например **template.html**. Ниже приведен пример того, как может выглядеть этот файл:

```
<!DOCTYPE html>
<html>
```

```
<head>
<title>{{ title }}</title>
</head>
<body>
<h1>Hello {{ name }}!</h1>
{% if messages %}
<ul>
{% for message in messages %}
<li>{{ message }}</li>
{% endfor %}
</ul>
{% else %}
<p>No messages.</p>
{% endif %}
</body>
</html>
```

В этом шаблоне заголовок `{{ title }}`, имя `{{ name }}` и сообщения `{{ messages }}` являются заполнителями, которые будут заполнены данными, переданными из представления Flask.

3. В представлении Flask для обслуживания этого HTML-файла используйте функцию `render_template()`, передавая переменные в качестве аргументов-ключей:

```
from flask import render_template
@app.route('/')
def index():
return render_template('template.html', title='Home Page',
name='John Doe', messages=['Hello', 'Hi', 'Hey'])
```

Такая настройка позволяет серверу отображать динамический контент на основе полученных данных, делая ваше приложение интерактивным и реагирующим на запросы пользователей.

Управление статическими файлами

Статические файлы — это компоненты вашего сайта, которые не изменяются в динамическом режиме. Например, файлы JavaScript, таблицы стилей CSS и изображения. Эффективное управление этими файлами имеет решающее значение для производительности и удобства обслуживания.

Организация статических файлов

Создайте в каталоге проекта папку **static**. Flask автоматически выполнит поиск статических файлов в папке с именем **static**. Для более четкой структуры следует разделить файлы на подпапки **css**, **js** и **images**.

```
/YourApp
/static
/css
style.css
/js
script.js
/images
logo.png
```

Обслуживание статических файлов

В Flask есть встроенный способ ссылки на статические файлы, находящиеся в ваших шаблонах. Используйте для генерации URL-адресов статических файлов функцию `url_for()`:

```
<link rel="stylesheet" type="text/css" href="
{{ url_for('static', filename='css/style.css')}}">
<script src="{{ url_for('static', filename='js/script.js') }}">
</script>
<img src="{{ url_for('static', filename='images/logo.png') }}"
alt="Company Logo">
```

Этот метод гарантирует, что для ваших файлов будут сгенерированы правильные пути, независимо от того, где размещено ваше приложение. Данный метод помогает избежать жесткого кодирования путей, которое может привести к ошибкам и усложнить поддержку вашего кода.

При этом соблюдайте следующие правила работы с шаблонами и статическими файлами.

- Поскольку изменение статических файлов происходит редко, вам следует внедрить методы кеширования, чтобы сократить время загрузки для постоянных посетителей. Обычно это можно настроить на уровне веб-сервера или с помощью HTTP-заголовков.

- Для рабочего режима минимизируйте размеры файлов CSS и JavaScript, чтобы сократить время загрузки и повысить пропускную способность.
- Используйте в Jinja2 функцию наследования шаблонов, чтобы создать базовый «скелетный» шаблон, содержащий все общие элементы вашего сайта (например, верхний и нижний колонтитулы и навигацию), и распространить этот базовый шаблон на другие шаблоны. Это уменьшает избыточность и помогает более эффективно управлять изменениями.

```
<!-- base.html -->
<html>
<head>
<title>{% block title %}{% endblock %}</title>
</head>
<body>
{% block body %}
{% endblock %}
</body>
</html>
<!-- home.html -->
{% extends "base.html" %}
{% block title %}Home{% endblock %}
{% block body %}
<h1>Welcome to the Home Page</h1>
<p>Hello, {{ name }}!</p>
{% endblock %}
```

Убедитесь, что ваше приложение Flask поддерживает аккуратный, привлекательный пользовательский интерфейс и предоставляет контент динамически, эффективно управляя статическими файлами и шаблонами. Помимо повышения эффективности, эти приемы помогут вашему веб-приложению выглядеть более стильно и профессионально, что, в свою очередь, улучшит взаимодействие с пользователями.

Работа с формами и загрузка файлов

Flask — работа с формами

Формы необходимы для сбора данных, вводимых пользователем. Flask может обрабатывать данные формы с помощью объекта `request`. Чтобы продемонстрировать работу с формами, мы используем в качестве примера простую контактную форму (или форму обратной связи).

Создание HTML-формы

В каталоге **templates** создайте новый файл с именем **contact.html**. Ниже приведена базовая форма для ввода данных пользователем:

```
<form method="post" action="{{url_for('handle_form') }}">
<label for="name">Name:</label>
<input type="text" id="name" name="name"><br><br>
<label for="email">Email:</label>
<input type="text" id="email" name="email"><br><br>
<input type="submit" value="Submit">
</form>
```

Данная форма собирает информацию об имени и электронной почте пользователя, которая будет отправлена в маршрут Flask для обработки.

Создание маршрута для отображения формы

В своем приложении определите маршрут Flask для отображения этой формы:

```
@app.route('/contact')
def contact():
return render_template('contact.html')
```

Обработка данных формы

После отправки формы данные должны быть собраны и обработаны по другому маршруту. Ниже показано, как можно обработать POST-запрос:

```
from flask import request, redirect, url_for
@app.route('/handle_form', methods=['POST'])
def handle_form():
name = request.form['name']
email = request.form['email']
# Обработать или сохранить данные формы
return redirect(url_for('thank_you'))
```

Этот код извлекает данные из полей формы, после чего эти данные можно обрабатывать по мере необходимости (например, сохранять в базе данных или отправлять по электронной почте).

Загрузка файлов

Управление загрузкой файлов позволяет пользователям отправлять файлы через формы. Это может быть особенно полезно для приложений, в которых требуется отправлять документы, изображения и другие медиафайлы.

Модификация HTML-формы для загрузки файлов

Чтобы разрешить загрузку файлов, измените форму, добавив в нее атрибут `enctype` и поле ввода для загружаемого файла:

```
<form method="post" action="{{url_for('handle_file_upload') }}"
enctype="multipart/form-data">
<label for="uploaded_file">Upload file:</label>
<input type="file" id="uploaded_file" name="file">
<input type="submit" value="Upload">
</form>
```

Атрибут `enctype="multipart/form-data"` (enctype=«составная_часть/форма-данные») необходим для того, чтобы указать браузеру, как правильно упаковать загружаемый файл.

Обработка загрузки файлов в Flask

Создайте маршрут для выполнения загрузки файлов. Необходимо убедиться, что файл можно безопасно сохранить, а затем обработать его соответствующим образом.

```
import os
from werkzeug.utils import secure_filename
```

```
@app.route('/handle_file_upload', methods=['POST'])
def handle_file_upload():
file = request.files['file']
if file and allowed_file(file.filename):
filename = secure_filename(file.filename)
file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
return redirect(url_for('uploaded_file', filename=filename))
return 'File upload unsuccessful'
```

В приведенном выше фрагменте кода `secure_filename` (безопасное_имя_файла) гарантирует, что имя файла не содержит вредоносных символов, а `allowed_file` (разрешенный файл) проверяет расширение файла на соответствие набору разрешенных расширений (функция, с помощью которой определяются безопасные типы файлов для загрузки).

При работе с формами и загрузками всегда проверяйте и обеззараживайте входящие данные, чтобы предотвратить такие распространенные уязвимости, как SQL-инъекции, межсайтовый скриптинг (XSS) и другие. Для работы с формами можно использовать расширение Flask-WTF, которое упрощает создание форм и выполняет валидацию. Установите максимальный размер файла для загрузки. Flask позволяет настроить конфигурацию `MAX_CONTENT_LENGTH` (максимальная длина содержимого) для отклонения входящих запросов со слишком большим объемом данных.

Основы интеграции баз данных

Любое динамическое веб-приложение на базе фреймворка Flask нуждается в постоянном хранении данных и должно быть интегрировано с базой данных. Flask рассчитан на работу с целым рядом технологий, обеспечивающих взаимодействие с базами данных. Для работы с базами данных используется язык структурированных запросов SQL. Также Flask работает с нереляционными базами данных NoSQL. Однако по умолчанию в Flask нет ни ORM, ни слоя абстракции баз данных. В этом разделе мы рассмотрим, как с помощью запросов SQL подключить базу данных к проекту Flask, воспользовавшись SQLAlchemy, известного объектно-реляционного связующего (ORM) для Python.

Примечание переводчика

SQL (Structured Query Language, или язык структурированных запросов) — это декларативный язык программирования (язык запросов), который используют для создания, обработки и хранения данных в реляционных БД. На чистом SQL нельзя написать программу — он предназначен только для взаимодействия с базами данных: получения, добавления, изменения и удаления информации в них, управления доступом и т. д.

NoSQL — это семейство нереляционных баз данных. В них разработчики отошли от использования традиционной табличной модели представления информации.

ORM (Object Relational Mapping, объектно-реляционное отображение) — это промежуточный уровень абстракции в серверной разработке, который действует как мост между языками объектно-ориентированного программирования (ООП) и реляционными базами данных.

Слой абстракции базы данных (Database abstraction layer, DBAL) — это интерфейс прикладного программирования, который унифицирует связь между компьютерным приложением и системами управления базами данных (СУБД), такими как SQL Server, DB2, MySQL, PostgreSQL, Oracle или SQLite.

Уровни абстракции уменьшают объем работы, предоставляя последовательный API разработчику и максимально скрывая специфику базы данных за этим интерфейсом.

Существует множество слоев абстракции с различными интерфейсами на многих языках программирования. Уровни абстракции базы данных:

- физический уровень (низший уровень);
- концептуальный или логический уровень (средний или высший уровень);
- уровень просмотра (внешний уровень).

SQLAlchemy — это программная библиотека на языке Python, предназначенная для работы с реляционными базами данных, в которых применены технологии ORM. Служит для синхронизации объектов Python и записей реляционной базы данных. ORM — комплекс программ, позволяющих работать с базами данных, как если бы они были объектами языка программирования, в данном случае Python.

Настройка SQLAlchemy с помощью Flask

SQLAlchemy предоставляет мощный и гибкий способ взаимодействия с базами данных через Pythonic-модели и запросы. Flask-SQLAlchemy — это расширение, которое поддерживает работу SQLAlchemy в вашем приложении на Flask в более удобном для Flask виде.

Установка Flask-SQLAlchemy

Сначала вам нужно добавить Flask-SQLAlchemy в свое окружение:

```
pip install flask_sqlalchemy
```

Настройка приложения

Добавьте настройки конфигурации базы данных в ваше приложение Flask:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///yourdatabase.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
```

В приведенном выше фрагменте кода `SQLALCHEMY_DATABASE_URI` используется для определения пути к файлу базы данных с указанием типа и местоположения базы данных.

Для `SQLALCHEMY_TRACK_MODIFICATIONS` определено значение `False`, чтобы выключить обработку сигналов. Это позволит экономить ресурсы.

Определение моделей

Модели в SQLAlchemy используются для определения структуры таблиц вашей базы данных на языке Python. Модели представляют собой классы, с помощью которых определяются поля таблицы, и могут включать методы для взаимодействия с данными.

```
class User(db.Model):
id = db.Column(db.Integer, primary_key=True)
```

```
username = db.Column(db.String(80), unique=True, nullable=False)
email = db.Column(db.String(120), unique=True, nullable=False)
def __repr__(self):
return '<User %r>' % self.username
```

В приведенном выше фрагменте кода `User` — это модель с полями `id`, `username` и `email`. Поле `id` определено как первичный ключ.

Создание базы данных

Определив модели, вы можете создать таблицы базы данных:

```
db.create_all()
```

Эта команда просматривает все классы, унаследованные от `db.Model`, и создает для них таблицы, если таковые еще не созданы.

Взаимодействие с базой данных

SQLAlchemy позволяет взаимодействовать с базой данных с помощью кода на Python, а не писать SQL-запросы, что помогает сохранить целостность и безопасность данных.

Вставка данных

```
new_user = User(username='johndoe', email='johndoe@gitforgits.com')
db.session.add(new_user)
db.session.commit()
```

Этот фрагмент кода создает новый экземпляр модели `User` и добавляет его в сессию базы данных. С помощью `db.session.commit()` фиксируются все незавершенные транзакции в базе данных.

Запрос данных

```
user = User.query.filter_by(username='johndoe').first()
print(user.email)
```

Этот запрос извлекает из базы данных информацию о первом пользователе с именем `'johndoe'` и выводит его электронную почту.

Обновление данных

```
user = User.query.filter_by(username='johndoe').first()
user.email = 'newemail@gitforgits.com'
db.session.commit()
```

В приведенном выше фрагменте кода обновляются данные об электронной почте пользователя, а изменения фиксируются в базе данных.

Удаление данных

```
user = User.query.get(1) # Assumes '1' is the id of the user
db.session.delete(user)
db.session.commit()
```

Этот фрагмент кода извлекает информацию о пользователе по его идентификатору, а затем удаляет его из базы данных.

Работа с отношениями

SQLAlchemy также может работать с отношениями между таблицами:

```
class Post(db.Model):
id = db.Column(db.Integer, primary_key=True)
title = db.Column(db.String(100), nullable=False)
content = db.Column(db.Text, nullable=False)
user_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)
user = db.relationship('User',
backref=db.backref('posts', lazy=True))
```

В приведенном выше примере отношение `user` позволяет нам легко получить доступ ко всем постам, созданным пользователем. Аргумент `backref` в методе `db.relationship()` создает обратное отношение, в котором вы можете получить доступ к постам из объекта `user`.

Извлечение связанных данных

Чтобы извлечь данные, использующие отношения между таблицами, можно сделать следующее:

```
user = User.query.get(1) # Извлечение данных пользователя с id=1
posts = user.posts # Доступ ко всем сообщениям,
                   # сделанным этим пользователем
for post in posts:
print(post.title)
```

Код демонстрирует, как извлечь данные пользователя по его ID, а затем через атрибут `posts` получить доступ ко всем сообщениям, связанным с этим пользователем. Этот атрибут обеспечивается параметром `backref='posts'` в отношениях, определенных в модели `Post`.

Использование опций запросов

SQLAlchemy предоставляет множество опций для оптимизации и настройки запросов, таких как ленивая загрузка, ускоренная загрузка и динамическая загрузка, которые могут помочь в управлении сеансами базы данных и производительностью.

- **Ленивая загрузка.** Поведение, установленное по умолчанию, при котором SQLAlchemy загружает данные в отдельном запросе для каждого доступа по мере необходимости. Подходит для небольших баз данных или малого количества отношений.
- **Ускоренная загрузка.** С помощью объединений и подзапросов загружает все данные сразу. Это полезно, когда вы знаете, что вам понадобятся все связанные данные, и хотите свести к минимуму количество запросов.
- **Динамическая загрузка.** В этом варианте отношения настраиваются как запросы, которые не загружаются до тех пор, пока не будет задан конкретный запрос. Это полезно для больших наборов связанных элементов.

Ниже описано, как можно задать ускоренную загрузку для оптимизации получения данных:

```
from sqlalchemy.orm import joinedload
user = User.query.options(joinedload(User.posts)).filter_by(username=
'johndoe').first()
for post in user.posts:
print(post.title)
```

При таком подходе SQLAlchemy настраивается на загрузку с помощью одного запроса всех связанных объектов `Post` одновременно с объектом `User`, что может значительно повысить производительность за счет сокращения количества обращений к базе данных.

Независимо от того, с какими операциями вы имеете дело — с основными или с расширенными транзакциями, SQLAlchemy обеспечит вас всеми необходимыми инструментами для безопасного и эффективного доступа к вашей базе данных. Таким образом, подключение к базе данных выполняет двойную функцию: улучшает функциональность и гарантирует, что ваши приложения Flask смогут без проблем и увеличиваться и уменьшаться.

Введение в расширения Flask

Со временем вы можете обнаружить, что встроенных возможностей Flask недостаточно для удовлетворения потребностей вашего приложения, поскольку оно становится все более сложным и функциональным. В этом случае становится целесообразным использование расширений Flask. Начиная с возможностей управления объектными отношениями (ORM) и библиотеки SQLAlchemy, о которых мы рассказывали ранее, и заканчивая аутентификацией пользователей, управлением миграцией и многим другим. Обширная экосистема расширений Flask может значительно расширить возможности ваших приложений.

Описание расширений Flask

Расширения Flask — это пакеты или модули, позволяющие дополнить функциональность Flask. Эти расширения разрабатываются для бесшовной интеграции с приложениями Flask, придерживаясь шаблонов и соглашений фреймворка. Данные расшире-

ния часто упрощают использование других библиотек в Flask, обрабатывают шаблонный код и добавляют новые возможности.

Рассмотрим часто используемые расширения Flask, которые позволяют решать различные задачи при разработке веб-приложений.

Flask-WTF

Упрощает обработку форм, интегрируя библиотеку WTForms, с помощью которой удобно управлять веб-формами. Библиотека содержит защиту CSRF и средства проверки подлинности, гарантирующие безопасность и достоверность предоставляемых пользователями данных.

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import InputRequired, Length, Email
class LoginForm(FlaskForm):
username = StringField('Username', validators=[InputRequired(),
Length(min=4, max=15)])
password = PasswordField('Password', validators=[InputRequired(),
Length(min=8, max=80)])
submit = SubmitField('Login')
```

Flask-SQLAlchemy

Создает слой ORM (объектно-реляционное отображение) для выполнения операций с базой данных в Flask и использованием программной библиотеки SQLAlchemy. Это упрощает манипулирование базой данных, делая ее более интуитивно понятной и поддерживаемой на Python.

```
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy(app)
```

Flask-Migrate

Служит для миграций баз данных SQLAlchemy для приложений Flask. В расширении Flask-Migrate используется инструмент для миграции базы данных Alembic, который позволяет отслеживать изменения схемы базы данных с помощью миграций, которые

могут быть общими и применяться к различным экземплярам приложения.

```
from flask_migrate import Migrate
migrate = Migrate(app, db)
```

> ***Примечание переводчика***
>
> Миграция базы данных — это процесс изменения структуры и содержимого базы данных с целью обновления ее версии, переноса на другую платформу или слияния с другой базой данных.

Flask-Login

Расширение обеспечивает управление сеансами пользователей и решает общие задачи по входу в систему, выходу из нее и запоминанию сеансов пользователей в течение длительного времени.

```
from flask_login import LoginManager, UserMixin, login_user,
logout_user, current_user
login_manager = LoginManager(app)
login_manager.login_view = 'login'
```

Flask-Mail

Добавляет в ваши приложения Flask возможность отправки электронной почты по протоколу SMTP, что удобно для таких функций, как письма с подтверждением пользователя, сброс пароля и уведомления.

```
from flask_mail import Mail, Message
mail = Mail(app)
```

Flask-RESTful

Поддерживает передовой опыт использования REST и упрощает создание REST API. Расширение обеспечивает ресурсо-ориентированный подход к созданию HTTP API, а также обрабатывает запросы и форматирует выходные данные.

```
from flask_restful import Api, Resource
api = Api(app)
class HelloWorld(Resource):
```

```
def get(self):
return {'hello': 'world'}
api.add_resource(HelloWorld, '/')
```

Использование расширений Flask

Чтобы использовать расширение Flask, как правило, нужно выполнить следующие шаги:

1. Установка расширения.

   Используйте менеджер пакетов pip для установки расширения.

   ```
   pip install flask-wtf
   ```

2. Импорт и инициализация.

   Импортируйте расширение и инициализируйте его с вашим приложением Flask.

   ```
   from flask_wtf import CSRFProtect
   app = Flask(__name__)
   csrf = CSRFProtect(app)
   ```

3. Настройка.

   Для работы многих расширений требуются определенные параметры конфигурации, которые обычно задаются непосредственно в файле конфигурации приложения Flask.

   ```
   app.config['SECRET_KEY'] = 'your_secret_key'
   app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///db.sqlite'
   ```

Затем вы просто включаете функционал расширения в логику своего приложения. Расширения повышают возможности Flask и позволяют создавать более структурированный код и приложения с большим количеством функций. По мере того как вы будете развивать свои навыки работы с Flask, использование различных расширений может существенно повлиять на ваш подход к созданию приложений, позволяя вам сосредоточиться на уникальной бизнес-логике, а не на общих функциональных возможностях внутренних компонентов.

Развертывание приложения Flask

После того как вы создали приложение Flask и интегрировали с помощью расширений многочисленные возможности, необходимо приложение развернуть, чтобы пользователи могли получить доступ к нему через Интернет. Подготовка проекта Flask к производству, выбор хостинг-провайдера, настройка веб-сервера и сервера приложений WSGI — все это является частью процесса развертывания.

Подготовка приложения Flask к развертыванию

- Переключение на производственную базу данных.

  Если вы при создании приложения использовали SQLite или другую базу данных, предназначенную для разработки, подумайте о переходе на базу данных, пригодную для эксплуатации в условиях производства. Например, PostgreSQL, MySQL или другую надежную систему управления базами данных, способную обеспечить одновременный доступ и работать с большими объемами данных.

- Использование переменных окружения для конфигурации.

  Конфиденциальную информацию, такую как URI базы данных, секретные ключи и учетные данные API сторонних разработчиков, следует хранить в переменных окружения, а не в исходном коде приложения. Это повышает безопасность и гибкость при работе в различных средах.

- Отключение режима отладки.

  Обязательно отключите режим отладки в вашем приложении Flask. Если в производственной среде оставить режим отладки включенным, это может привести к раскрытию конфиденциальной информации и появлению уязвимостей.

```
app.config['DEBUG'] = False
```

Выбор хостинга

Для размещения приложений, созданных на платформе Flask, существует несколько вариантов — от традиционных виртуальных хостингов до самых современных платформ для приложений.

- **Heroku.** Это популярная «платформа-как-сервис» (PaaS), облегчающая развертывание, управление и масштабирование веб-приложений. Heroku может запускать приложения, созданные на основе Flask, и выполняет большую часть настроек автоматически.
- **DigitalOcean, AWS и Google Cloud.** Эти провайдеры «инфраструктура как услуга» (IaaS) предлагают дополнительный контроль над сервером и средой. Здесь требуется немного больше настроек, но в этом случае обеспечивается гибкость в настройке системы под ваши конкретные нужды.
- **PythonAnywhere.** Онлайн IDE и хостинг. Данный сервис особенно хорошо подходит для приложений, написанных на языке Python с использованием фреймворка Flask.

Настройка веб-сервера и сервера приложений WSGI

Чтобы предоставить пользователям ваше приложение на основе фреймворка Flask, вам понадобится веб-сервер и сервер приложений WSGI (Web Server Gateway Interface). Веб-сервер обрабатывает HTTP-запросы и обслуживает статические файлы, а сервер WSGI выполняет ваш Python-код.

- **Gunicorn.** Это популярный WSGI-сервер для систем на базе Unix. Данный сервер легкий, поддерживает несколько рабочих процессов и легко настраивается с помощью Flask.
- **Nginx.** Надежный, высокопроизводительный веб-сервер, который может работать в качестве обратного прокси и балансировщика нагрузки. Обычно он используется в сочетании с Gunicorn для обслуживания статических файлов и обработки клиентских соединений.

Использование Heroku

Развертывание приложения Flask на сервере Heroku включает в себя следующие шаги.

Подготовка приложения

Убедитесь, что структура приложения соответствует требованиям, а все зависимости перечислены в файле **requirements.txt**.

Создайте модуль профилирования `Procfile`, который указывает серверу Heroku, как выполнять ваше приложение:

```
web: gunicorn app:app
```

В этом модуле профилирования `web` указывает, что данный тип процесса является доступным через веб, а `gunicorn app:app` определяет, что Heroku должен использовать Gunicorn для запуска приложения.

Настройка Git-репозитория

Инициализируйте Git-репозиторий в папке проекта и зафиксируйте файлы, если вы этого еще не сделали.

```
git init
git add .
git commit -m "Initial commit"
```

Создание приложения Heroku

Установите Heroku CLI и войдите в свою учетную запись Heroku. Создайте новое приложение:

```
heroku create your-app-name
```

Разверните приложение, разместив его код на Heroku:

```
git push heroku master
```

Затем установите переменные окружения на Heroku:

```
heroku config:set FLASK_APP=run.py
heroku config:set SECRET_KEY='your_secret_key'
```

Выполнение проверки развертывания

Запустите приложение в веб-браузере, используя URL-адрес, предоставленный Heroku, или запустив его на выполнение следующей командой:

```
heroku open
```

Перед развертыванием приложения Flask необходимо выполнить ряд важных действий, таких как подготовка приложения к производству, выбор хорошего хостинг-провайдера, настройка веб-сервера и сервера приложений WSGI. Если вы будете следовать этим шагам, ваше приложение Flask будет развернуто безопасно и эффективно, оно будет готово работать с реальным трафиком и взаимодействием.

Резюме

В этой главе мы рассмотрели базовые принципы создания и выпуска веб-приложений, созданных на основе фреймворка Flask. На первом этапе знакомства с Flask мы познакомились с базовыми понятиями, такими как создание базовой среды Flask и создание веб-приложений с минимальным количеством кода, но достаточной мощностью. После того как мы освоили маршрутизацию, следующим шагом перешли к представлениям, которые необходимы для определения того, как приложение реагирует на пользовательский ввод.

Кроме того, мы узнали, как фреймворк Flask работает со статическими файлами и шаблонами. Мы смогли легко интегрировать логику Python в наши веб-страницы с помощью шаблонов, что позволило нам динамически генерировать HTML-контент с помощью шаблонизатора Jinja2. Функциональность и эстетика приложения дополняются статическими файлами, которые необходимо правильно обрабатывать, используя JavaScript, CSS и файлы изображений. Безопасный сбор и обработка пользовательских данных — важнейшая функция интерактивных сайтов, и в этой главе мы узнали, как это сделать, ознакомившись с обработкой форм и загрузкой файлов. Пакет Flask-SQLAlchemy позволил нам подключиться к базе данных и эффективно выполнять действия

CRUD (Create, Read, Update и Delete или создание, чтение, обновление и удаление).

В заключение мы познакомились с несколькими расширениями Flask, предоставляющими дополнительные возможности, такие как аутентификация пользователей, обработка электронной почты и расширенные возможности управления объектными отношениями. Завершением главы стал полный практический курс по запуску приложения Flask. В целях обеспечения готовности программы к производству мы протестировали несколько методов развертывания и создали окружения. В этой главе по мере создания веб-приложения с помощью фреймворка Flask с нуля мы узнали обо всем: от базовой маршрутизации до продвинутого взаимодействия с базами данных и тактики развертывания.

ГЛАВА 3
Дополнительные возможности Flask

Введение

В этой главе рассматриваются дополнительные возможности Flask. К дополнительным возможностям относится и применение шаблонов, и использование передового опыта. Эти дополнительные возможности позволят обеспечить безопасность и масштабирование приложений, созданных на основе фреймворка Flask. При увеличении размера и сложности проектов на основе Flask возникают новые проблемы, для решения которых требуется применение более сложных подходов и методов. Если вы хотите максимально повысить производительность, надежность, безопасность и усовершенствовать структуру больших приложений, эта глава для вас.

Сначала мы узнаем, как для создания структуры больших приложений использовать модуль Flask Blueprints, что в переводе значит чертёж или эскиз. Использование эскизов облегчает управление и масштабирование огромных проектов, разбивая их на более мелкие, многократно используемые компоненты. Далее мы рассмотрим шаблон Flask Application Factory, с помощью которого удобно управлять различными настройками и окружениями приложения, поскольку этот шаблон универсален и настраивается при создании экземпляров Flask.

После знакомства с RESTful-сервисами мы рассмотрим, как использовать Flask-RESTful — расширение, которое упрощает процесс разработки REST API. Как создавать и разворачивать масштабируемые RESTful API — тема данного раздела.

Защита веб-приложений требует надежной аутентификации и авторизации пользователей. Здесь вы узнаете о нескольких под-

ходах к аутентификации, управлению сеансами и о контроле доступа на основе разрешений в приложениях.

Надежность и возможность контроля приложений также существенно зависят от методов обработки ошибок и протоколирования. Мы расскажем о некоторых методах обработки и документирования исключений и ошибок, а также о некоторых подходах, применяемых при отладке и устранению неполадок.

Далее разговор коснется стратегий оптимизации производительности с акцентом на то, как сделать Flask-приложения более быстрыми и эффективными. Здесь мы рассмотрим несколько методов улучшения работы приложений на основе Flask, включая то, как взаимодействовать с базами данных и обрабатывать запросы.

Создание модульной структуры больших приложений с помощью Flask Blueprints

По мере роста сложности и размера приложений Flask управление всем приложением в рамках одного модуля или скрипта становится нецелесообразным. Именно в этом случае становится полезной функция Flask Blueprints. Создаваемые с ее помощью эскизы позволяют разделить ваше приложение Flask на более мелкие, многократно используемые части, каждая из которых может работать как отдельное приложение. С помощью таких частей — эскизов можно сделать большие приложения проще и функциональнее.

Для чего предназначен Flask Blueprints?

Эскизы можно рассматривать как мини-приложения, которые не запускаются сами по себе, но во время выполнения регистрируются в основном приложении Flask. Эти мини-приложения позволяют «разбить» проект по функциональным возможностям и могут быть повторно использованы в нескольких проектах.

Эскизы облегчают процесс разграничения задач, позволяя изолировать функции и компоненты в отдельные части приложения, каждая из которых имеет свои статические файлы, шаблоны, представления, формы и другие элементы. Например, у вас может быть один эскиз для аутентификации, другой — для внутренней структуры блога, а третий — для интерфейса администратора.

Создание и регистрация эскизов

Эскиз сначала нужно создать, или определить, а потом зарегистрировать в приложении.

Определение эскиза

Начните с импорта класса `Blueprint`, а затем создайте его экземпляр. Конструктор `Blueprint` принимает два необходимых аргумента: имя эскиза и модуль или пакет, в котором находится эскиз.

```
from flask import Blueprint
auth = Blueprint('auth', __name__, template_folder='templates')
```

В приведенном выше фрагменте кода `auth` — это имя эскиза, а `__name__` гарантирует, что эскиз «понимает», в каком месте он определен. Аргумент `template_folder` является необязательным и может использоваться для указания места хранения шаблонов, применяемых в этом эскизе.

Регистрация эскиза

После определения эскиза, чтобы его использовать, его следует зарегистрировать в приложении. Обычно это делается в вашей фабрике веб-приложений.

> ***Примечание переводчика***
>
> *Фабрика приложений (ФП)* — это автоматизированная среда, ориентированная на максимальную автоматизацию производства продукта. В сфере информационных технологий концепция фабрики приложений ориентирована на переход к индустриально ориентированному производству программного обеспечения. ФП ускоряет процесс создания программных компонентов, приложений или систем за счет предоставления специалистам руководств

и рекомендаций, в которых устанавливается последовательность действий.

Главенствующую роль играет формализация процесса:

- компоненты, которые можно быстро собрать и настроить;
- прототипы систем, которые можно быстро подготовить;
- специализированные средства, полностью или частично автоматизирующие механические рутинные задачи.

```
from flask import Flask
from yourapplication.auth import auth
app = Flask(__name__)
app.register_blueprint(auth, url_prefix='/auth')
```

Параметр `url_prefix` необязателен, но удобен при добавлении префикса ко всем маршрутам, зарегистрированным в эскизе. В данном случае все маршруты, определенные в эскизе `auth`, будут иметь префикс `/auth`.

Структурирование представлений в эскизах

После настройки эскиза можно приступать к добавлению в него маршрутов и представлений. Ниже показано, как добавить представление в эскиз `auth`.

```
# в вашем приложении /auth/views.py)
from . import auth
@auth.route('/login', methods=['GET', 'POST'])
def login():
return "Login Here"
```

Использование эскизов

Когда ваше приложение разрастется, у вас может появится несколько отдельных функций, каждая из которых потенциально заслуживает собственного эскиза. Ниже описано, как можно структурировать большое приложение.

```
/yourapplication
/auth
__init__.py
```

```
views.py
models.py
/blog
__init__.py
views.py
models.py
/admin
__init__.py
views.py
/static
/templates
__init__.py
config.py
run.py
```

В этой структуре:

- для каждой функции (`auth`, `blog`, `admin`) создана отдельная директория, со своими шаблонами и статическими файлами;
- в корне каталога каждой функции находится файл `__init__.py`, в котором функция определяется и конфигурируется.

Эскизы улучшают масштабируемость приложений Flask, позволяя разработчикам объединять функции в отдельные компоненты, что упрощает управление и расширение кодовой базы. Модульная архитектура позволяет улучшить производственные отношения в больших командах разработчиков, сводя к минимуму программные конфликты при слиянии и обеспечивая независимое развитие различных частей приложения, которые могут развиваться независимо друг от друга.

Шаблон Application Factory во Flask

Описание шаблона Application Factory

Использование шаблона Flask Application Factory значительно повышает масштабируемость и гибкость приложений Flask, особенно по мере увеличения размера и сложности приложения. В шаблоне Application Factory (Фабрика приложений) определяется функция, которая создает и возвращает экземпляр приложе-

ния Flask с различными конфигурациями. Основное преимущество данного шаблона заключается в том, что он позволяет динамически изменять конфигурацию, расширения, эскизы и другие компоненты в зависимости от среды выполнения или других внешних факторов. При этом не требуется изменять исходный код приложения, что повышает удобство обслуживания и гибкость развертывания.

Использование шаблона Application Factory

Ниже приводится пошаговая процедура по применению шаблона Flask Application Factory.

Создание функции *Factory*

Функция `factory` (фабрика), как правило, находится в базовом пакете вашего приложения. Эта функция настраивает и возвращает объект приложения Flask.

```
from flask import Flask
from config import Config
def create_app(config_class=Config):
app = Flask(__name__)
app.config.from_object(config_class)
return app
```

В приведенном выше примере кода `create_app` принимает в качестве аргумента класс конфигурации, который используется для настройки экземпляра приложения Flask. По умолчанию классом `config_class` является `Config`, который должен быть классом Python, содержащим переменные конфигурации.

Настройка параметров конфигурации

Параметры конфигурации следует организовать в различные классы в отдельном файле **config.py**. Каждый класс должен соответствовать разным средам развертывания.

```
class Config:
DEBUG = False
TESTING = False
DATABASE_URI = 'sqlite:///example.db'
```

```
class DevelopmentConfig(Config):
DEBUG = True
class TestingConfig(Config):
TESTING = True
DATABASE_URI = 'sqlite:///test.db'
class ProductionConfig(Config):
DATABASE_URI = 'mysql://user@localhost/foo'
```

Такая настройка позволяет легко изменять конфигурацию, передавая различные классы конфигурации в функцию `create_app`.

Регистрация сценариев и расширений

Внутри функции `factory` после настройки приложения необходимо зарегистрировать все расширения и эскизы. Этот шаг гарантирует, что данные компоненты будут привязаны именно к этому экземпляру приложения.

```
from .extensions import db, migrate
from .routes import main, admin
def create_app(config_class=Config):
app = Flask(__name__)
app.config.from_object(config_class)
db.init_app(app)
migrate.init_app(app, db)
app.register_blueprint(main)
app.register_blueprint(admin)
return app
```

В приведенном выше фрагменте кода `db` и `migrate` являются расширениями Flask, а `main` и `admin` — эскизами.

На практике при локальном запуске вы будете использовать шаблон `factory`:

```
export FLASK_APP=myapp:create_app('DevelopmentConfig')
flask run
```

И в производстве:

```
export FLASK_APP=myapp:create_app('ProductionConfig')
```

Этот метод позволяет переменной окружения `FLASK_APP` указать не только приложение для запуска, но и то, как оно должно быть

настроено. Функция `factory create_app` является мощной и универсальной, позволяя вам адаптировать экземпляр Flask к потребностям каждого конкретного окружения или ситуации.

Реализация RESTful-сервисов с помощью Flask-RESTful

RESTful-сервисы — это технология разработки веб-приложений, взаимодействующих по протоколу HTTP в соответствии с основными протоколами Интернета. Принципы REST (Representational State Transfer или передача данных о состоянии представления) позволяют обеспечить масштабируемое и простое взаимодействие за счет использования стандартных методов HTTP, таких как GET, POST, PUT, DELETE и т. д., для выполнения действий. Эти принципы помогают создавать сервисы, которые не имеют статических данных и могут быть легко использованы различными клиентами, включая браузеры, мобильные приложения и другие веб-сервисы.

Введение в Flask-RESTful

Flask-RESTful — это расширение для Flask, позволяющее быстро создавать REST API. Это простая абстракция, которая работает поверх обычных механизмов маршрутизации Flask. Flask-RESTful упрощает создание API благодаря ресурсоориентированному подходу, где каждый ресурс соответствует определенной сущности и может быть доступен или управляем с помощью стандартных методов HTTP.

> ***Примечание переводчика***
>
> Абстракция данных в объектно-ориентированном программировании — это только использование определения характеристик объекта, без описания их конкретных реализаций. Основная идея состоит в том, чтобы представить объект, обладающий набором методов, и при этом не предоставлять конкретную логику этих методов. Простыми словами, абстракция отвечает на вопрос «Что это?», без ответа на вопрос «Как это делается?».

Абстракция позволяет:

- Упростить сложность. Разработчики могут скрыть детали реализации и сосредоточиться только на ключевых аспектах системы.
- Разбить систему на модули или классы, которые могут работать независимо друг от друга. Это способствует повторному использованию кода и улучшает масштабируемость проекта.
- Повысить безопасность. Внешние компоненты не имеют прямого доступа к внутренним деталям объекта или системы.

Функционирование Flask-RESTful

Flask-RESTful предоставляет класс `Resource`, который, чтобы определить логику работы с различными HTTP-методами, следует разделить на подклассы. Связь между HTTP-методами и методами ресурса прямая: например, GET-запрос к ресурсу обрабатывается методом `get()` этого ресурса, POST-запрос — методом `post()` и т. д. Затем для связи с конечной точкой этот ресурс регистрируется в объекте API.

Создание REST API с помощью Flask-RESTful

Чтобы продемонстрировать создание REST API с помощью расширения Flask-RESTful, давайте создадим простой API для управления товарами в магазине.

1. Установите Flask-RESTful.

   Начните с установки расширения Flask-RESTful:

   ```
   pip install flask-restful
   ```

2. Настройте приложение Flask.

   Создайте новое приложение Flask и импортируйте расширение Flask-RESTful:

   ```
   from flask import Flask
   from flask_restful import Api, Resource
   app = Flask(__name__)
   api = Api(app)
   ```

3. Определите классы ресурсов.

   Определите класс ресурсов для вашей сущности. В данном случае создайте ресурс `Item` с методами для обработки различных методов HTTP:

```
class Item(Resource):
def get(self, name):
return {'item': name}
def post(self, name):
return {'item': name}, 201
def delete(self, name):
return {'message': f'Item {name} deleted'}
class ItemList(Resource):
def get(self):
return {'items': ['Item1', 'Item2']}
```

   Каждый метод соответствует методу RESTful HTTP. Например, `get` возвращает элемент, `post` создает элемент, а `delete` удаляет элемент.

4. Добавьте ресурсы в API.

   Зарегистрируйте классы ресурсов с определенными маршрутами:

```
api.add_resource(Item, '/item/<string:name>')
api.add_resource(ItemList, '/items')
```

   Метод `add_resource` связывает класс `Item` с маршрутом `/item/<string:name>`, где `<string:name>` — переменная часть URL.

5. Запустите приложение Flask.

   Настройте приложение Flask на запуск с включенной отладкой. Обычно это делается в условном блоке, чтобы предотвратить запуск, когда скрипт импортируется как модуль:

```
if __name__ == '__main__':
app.run(debug=True)
```

   В результате запускается локальный сервер разработки для приложения Flask.

Тестирование API

После запуска приложения Flask следует протестировать конечные точки API. Тестирование выполняется с помощью таких инструментов, как Postman или curl.

Например, чтобы получить список товаров, вы можете отправить GET-запрос на

http://localhost:5000/items,

а чтобы добавить товар, отправьте POST-запрос на

http://localhost:5000/item/apple.

Определив данные ресурсы и сопоставив их с маршрутами, вы сможете создать интуитивно понятный и масштабируемый API, соответствующий принципам REST. Таким образом, расширение Flask-RESTful является ценным инструментом для разработчиков, желающих создать эффективные и действенные REST API на Python с использованием Flask.

Аутентификация и авторизация пользователей

Аутентификация и авторизация являются гарантией того, что только авторизованные пользователи могут получить доступ к определенным ресурсам или выполнить определенные действия в соответствии со своими правами. Аутентификация и авторизация пользователей являются важнейшими компонентами безопасности веб-приложений. Если авторизация определяет, что может делать аутентифицированный пользователь, то аутентификация проверяет его личность.

Значение аутентификации, авторизации и управления сессиями

Аутентификация

Аутентификация подтверждает, что пользователи являются теми, за кого себя выдают.

Аутентификация гарантирует, что пользователи могут получить доступ к ресурсам только после успешного прохождения проверки подлинности, что является гарантией защиты конфиденциальной информации от несанкционированного доступа.

Авторизация

После прохождения аутентификации проверка авторизации гарантирует, что у пользователя есть правильные разрешения на доступ к ресурсу или выполнение определенной операции.

Внедряя подробные уровни разрешений, разработчики приложений сводят к минимуму риск случайного или злонамеренного использования системы.

Управление сессиями

Веб-приложениям часто требуется сохранять состояние пользователя (или сеанс) между различными запросами. Правильное управление сеансами обеспечивает безопасное сохранение этого состояния.

Реализация аутентификации в Flask

Flask не имеет встроенных механизмов аутентификации, но их можно легко реализовать с помощью расширений, таких как Flask-Login, предназначенных для работы с пользовательскими сессиями.

Установка Flask-Login

```
pip install flask-login
```

Настройка Flask-Login

Flask-Login обеспечивает управление пользовательскими сессиями в Flask. Расширение решает общие задачи по входу в систему, выходу из нее и запоминанию сеансов пользователей.

```
from flask_login import LoginManager, UserMixin, login_user,
logout_user, login_required
app = Flask(__name__)
```

```
app.secret_key = 'your_secret_key' # Для защиты сессий
login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view = 'login'
```

Функция загрузчика пользователя

Для `login_manager` требуется функция `user loader` (загрузка пользователя), которая загружает пользователя по заданному идентификатору.

```
@login_manager.user_loader
def load_user(user_id):
return User.query.get(int(user_id))
```

Определение модели пользователя

Модель пользователя должна наследоваться от класса `UserMixin`, который добавляет реализации по умолчанию для нескольких свойств и методов, используемых расширением Flask-Login.

```
from flask_sqlalchemy import SQLAlchemy
from flask_login import UserMixin
db = SQLAlchemy(app)
class User(db.Model, UserMixin):
id = db.Column(db.Integer, primary_key=True)
username = db.Column(db.String(25), unique=True, nullable=False)
password = db.Column(db.String(80), nullable=False)
```

Создание маршрутов аутентификации

Создайте маршруты для функций входа и выхода из системы.

```
@app.route('/login', methods=['GET', 'POST'])
def login():
if request.method == 'POST':
username = request.form['username']
password = request.form['password']
user =User.query.filter_by(username=username).first()
if user and user.password == password: # Simple password check
login_user(user)
return redirect(url_for('dashboard'))
```

```
else:
return 'Invalid username or password'
return render_template('login.html')
@app.route('/logout')
@login_required
def logout():
logout_user()
return redirect(url_for('login'))
@app.route('/dashboard')
@login_required
def dashboard():
return 'Welcome to your Dashboard!'
```

Примечание переводчика

Dashboard — *дашборд* или *информационная панель* — это интерактивная аналитическая панель, графический интерфейс. Смысл в том, что на одном экране отображаются все ключевые метрики, показатели цели или выполняемых процессов. С помощью этих метрик можно выявить и проанализировать тренды и изменения. Мы сталкиваемся с информационными панелями (дашбордами) каждый день. Приборная панель в автомобиле или графики активности в приложении фитнес-браслета — всё это дашборды или инфопанели.

Выполнение авторизации

Для управления авторизацией следует расширить модель пользователя или использовать расширения Flask для работы с ролями и разрешениями.

Управление доступом на основе ролей (RBAC)

Вы можете указать роли и разрешения непосредственно в модели базы данных.

```
class Role(db.Model):
id = db.Column(db.Integer, primary_key=True)
name = db.Column(db.String(50), unique=True)
class UserRole(db.Model):
user_id = db.Column(db.Integer, db.ForeignKey('user.id'),
primary_key=True)
```

```
role_id = db.Column(db.Integer, db.ForeignKey('role.id'),
primary_key=True)
```

Проверка прав доступа

Декораторы или вспомогательные функции могут использоваться для проверки наличия у пользователя определенной роли или разрешения, прежде чем разрешить ему доступ к маршруту.

```
def requires_roles(*roles):
def wrapper(f):
@wraps(f)
def wrapped(*args, **kwargs):
if not current_user.has_role(roles):
return 'Unauthorized', 403
return f(*args, **kwargs)
return wrapped
return wrapper
@app.route('/admin')
@login_required
@requires_roles('admin')
def admin():
return 'Admin Page'
```

Приложения Flask могут защищать информацию о пользователях и обеспечивать доступ к ним только с помощью систем аутентификации и авторизации. Более того, данная настройка помогает поддерживать безопасную среду взаимодействия с конкретным пользователем, что, в свою очередь, защищает программу от нелегального доступа.

Обработка ошибок и ведение журнала

Чтобы облегчить устранение проблем в разрабатываемых приложениях, улучшить информативность для пользователей сообщений об ошибках, легко отслеживать действия программы для их анализа разработчиками, обязательно обрабатывать ошибки и протоколировать данные. Flask позволяет эффективно настраивать обработку ошибок и ведение журнала, чтобы контролиро-

вать перехват исключений и запись в журнал информации о ходе выполнения приложения.

Flask — обработка ошибок

Flask предоставляет встроенную поддержку обработки ошибок, возникающих в приложении, включая необработанные исключения и HTTP-ошибки. Ниже описано, как настроить пользовательскую обработку ошибок в приложении Flask.

Обработка ошибок приложения

Пользовательские обработчики ошибок в Flask можно указать с помощью декоратора `@app.errorhandler`, который перехватывает ошибки и исключения, возникающие во время выполнения.

```
@app.errorhandler(404)
def not_found_error(error):
return render_template('404.html'), 404
@app.errorhandler(500)
def internal_error(error):
db.session.rollback() # при условии, что это вы используете
                                                    SQLAlchemy
return render_template('500.html'), 500
```

В данном случае `404` и `500` — это коды состояния «Не найдено» и «Внутренняя ошибка сервера» соответственно. Функции возвращают пользовательские шаблоны, которые информируют пользователя об ошибке, используя более дружелюбную форму.

Обработка исключений

Flask позволяет обрабатывать исключения Python напрямую, обеспечивая контроль над реакцией, когда в логике приложения что-то идет не так.

```
@app.errorhandler(Exception)
def handle_exception(e):
# прохождение ошибок HTTP
if isinstance(e, HTTPException):
return e
```

```
# теперь вы обрабатываете только не-HTTP исключения
return render_template("error_generic.html", error=e), 500
```

Flask — ведение журнала

Мониторинг приложений и устранение неполадок в значительной степени зависят от регистрации ошибок в журнале. Для вывода сообщений в файл или на консоль Flask использует встроенный в Python модуль логирования. Чтобы отслеживать запросы, сбои и оперативные данные, настройте ведение журнала в проекте Flask, как показано ниже:

```
import logging
from logging.handlers import RotatingFileHandler
if not app.debug:
file_handler = RotatingFileHandler('instance/flask_app.log',
maxBytes=10240, backupCount=10)
file_handler.setFormatter(logging.Formatter(
'%(asctime)s %(levelname)s: %(message)s
[in %(pathname)s:%(lineno)d]'
))
app.logger.addHandler(file_handler)
app.logger.setLevel(logging.INFO)
app.logger.info('Flask application startup')
```

Эта конфигурация устанавливает обработчик «ротируемого» или циклически запускаемого журнала, что означает, что файл журнала будет автоматически «ротироваться» (т. е. запускаться процесс создания нового файла), когда текущий файл достигнет определенного размера. Обработчик также форматирует сообщения журнала, включая временные метки, уровень серьезности событий и источник записи в журнале.

Использование протоколирования для данных запроса

Чтобы выполнить отладку проблем, связанных с конкретными запросами, вам нужно регистрировать данные о каждом запросе и ответе. При правильной настройке Flask может автоматически регистрировать эту информацию.

```
from flask import request
@app.before_request
def log_request_info():
app.logger.debug('Headers: %s', request.headers)
app.logger.debug('Body: %s', request.get_data())
```

Эта функция регистрирует заголовки (`Headers`) и тело (`Body`) каждого входящего запроса, что может быть очень полезно для отладки проблем, связанных с конкретными HTTP-запросами.

Пользовательские средства регистрации

Работая с большими приложениями или приложениями с более высокими требованиями к ведению журналов, вы, возможно, захотите для различных частей вашего приложения установить собственные отдельные регистраторы:

```
custom_logger = logging.getLogger('custom_logger')
file_handler = logging.FileHandler('logs/custom.log')
file_handler.setLevel(logging.INFO)
file_handler.setFormatter(logging.Formatter(
'%(asctime)s - %(name)s - %(levelname)s - %(message)s'
))
custom_logger.addHandler(file_handler)
@app.route('/')
def index():
custom_logger.info('Index page is accessed.')
return "Hello, World!"
```

Такая настройка позволяет вести адресный журнал, не загрязняя основной журнал приложения слишком большим количеством подробных данных.

Методы оптимизации производительности

Чтобы ваше веб-приложение работало без сбоев и выдерживало нагрузки даже при резком увеличении трафика, необходимо в Flask оптимизировать производительность этого приложения. Мы будем оптимизировать работу приложений на основе Flask, использовав несколько стратегий, которые могут быть полезны,

в частности, для взаимодействия с базами данных и обработки запросов.

Оптимизация баз данных

Оптимизация запросов

Вместо применения функции SELECT * укажите только те столбцы, которые вам нужны. Такая настройка сократит объем данных, передаваемых, обрабатываемых и используемых в вашем приложении.

```
from models import User
users = User.query.with_entities(User.name, User.email).all()
```

Убедитесь, что столбцы, используемые в условиях WHERE, ORDER BY и JOIN, проиндексированы. Индексация может значительно ускорить выполнение запроса за счет уменьшения количества записей, которые необходимо просканировать в базе данных.

Пакетные вставки и обновления

При вставке или обновлении нескольких строк не выполняйте по одному запросу на строку, а объединяйте все строки в одну операцию. Это сокращает количество обращений к базе данных.

```
from app import db
from models import User
users = [User(name="User1"), User(name="User2"), User(name="User3")]
db.session.add_all(users)
db.session.commit()
```

Вы также можете воспользоваться пулом соединений, чтобы повторно использовать существующие соединения с базой данных, а не открывать новое соединение для каждого запроса. Flask-SQLAlchemy пул соединений обрабатывает автоматически.

Оптимизация обработки запросов

Эффективная последовательность данных

Если вашему приложению необходимо отправлять или получать большие объемы данных, оптимизируйте процесс их сериализа-

ции и десериализации. В случае с данными JSON ускоренную сериализацию и десериализацию данных по сравнению со встроенным в Python модулем `json` обеспечивают такие библиотеки, как `ujson` или `orjson`.

```
import orjson
def default(obj):
if isinstance(obj, Decimal):
return str(obj)
raise TypeError
json_data = orjson.dumps(data, default=default)
```

Примечание переводчика

Сериализация данных — это процесс перевода структуры данных в битовую последовательность. Обратной к операции сериализации является операция десериализации (структуризации) — создание структуры данных из битовой последовательности.

Асинхронные обработчики

Для операций, связанных с вводом-выводом, рекомендуется использовать асинхронные обработчики запросов. Flask изначально не поддерживает асинхронные представления, но вы можете интегрировать эти возможности с помощью серверов наподобие **gunicorn**, библиотек, таких как **gevent**, или использовать альтернативы Flask, например **Quart**, который поддерживает работу модуля asyncio.

```
# Использование gunicorn вместе с gevent
# gunicorn myapp:app -k gevent -workerconnections 1000
```

Примечание переводчика

Gunicorn — это сервер приложений, предназначенный для работы Web-приложений написанных на Python. Основная его задача — это работа в режиме демона и поддержка постоянной работы Web-приложений. Демон — это программа, которая запускается в фоновом режиме (без терминала или пользовательского интерфейса), ожидая событий и предлагая какие-то службы для их выполнения. Хорошим примером демона может служить веб-сервер, ожидающий запрос на доставку страницы, или сервер SSH, ожидающий чьего-нибудь логина. Работа демонов не видна.

Gevent — это библиотека для организации одновременных вычислений на основе libev. Gevent предоставляет удобный API для

задач, связанных с одновременной обработкой данных и работой с сетью.

Название модуля asyncio — это сокращение от asynchronous I/O (асинхронный ввод/вывод). Это Python-библиотека, которая позволяет выполнять код, используя модель асинхронного программирования.

Кеширование ответов

Реализуйте механизмы кеширования для хранения результатов сложных операций, чтобы последующие запросы обслуживались быстрее. Для кеширования представлений или данных можно использовать расширение Flask-Caching.

```
from flask_caching import Cache
cache = Cache(config={'CACHE_TYPE': 'SimpleCache'})
cache.init_app(app)
@app.route('/expensive_data')
@cache.cached(timeout=50)
def get_expensive_data():
data = calculate_expensive_data()
return jsonify(data)
```

Настройка приложений и веб-серверов

Для обработки клиентских соединений и статических файлов используйте обратный прокси-сервер, например Nginx. С помощью Nginx можно кешировать ответы, балансировать нагрузку между несколькими экземплярами приложения и управлять медленными клиентскими соединениями — все это поможет снизить нагрузку на ваше приложение Flask.

```
location /static {
alias /path/to/your/application/static;
}
location / {
proxy_pass http://localhost:8000;
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
}
```

Примечание переводчика

Обратный прокси-сервер (reverse proxy) — это тип прокси-сервера, который ретранслирует запросы клиентов из внешней сети на один или несколько серверов, логически расположенных во внутренней сети. При этом для клиента это выглядит так, будто запрашиваемые ресурсы находятся непосредственно на прокси-сервере.

Мониторинг и профилирование

Для мониторинга производительности приложения используйте такие инструменты, как New Relic или Prometheus. Эти инструменты помогут выявить узкие места и области, нуждающиеся в оптимизации. Составьте профиль приложения Flask, чтобы выявить наиболее ресурсоемкие части. Для профилирования можно использовать встроенный в Python модуль `cProfile` или инструменты типа `py-spy`.

```
import cProfile
import pstats
profiler = cProfile.Profile()
profiler.enable()
# запустите ваш код здесь
profiler.disable()
stats = pstats.Stats(profiler).sort_stats('cumtime')
stats.print_stats()
```

Благодаря этим подходам к оптимизации вы сможете сделать ваше приложение Flask быстрее за счет повышения его производительности. Если вы последуете этим рекомендациям, то убедитесь, что ваше программное обеспечение сможет обслуживать больше пользователей и улучшить их производительность.

Примечание переводчика

New Relic — это облачная платформа мониторинга и анализа производительности приложений и инфраструктуры.

Она предоставляет инструменты для:

- отслеживания и анализа работы приложений в реальном времени;
- обнаружения проблем производительности;
- оптимизации работы системы.

Основные возможности платформы New Relic следующие:

- мониторинг приложений;
- инфраструктурный мониторинг;
- анализ данных;
- диагностика и отладка проблем производительности;
- интеграция и расширяемость.

Prometheus — это бесплатная система серверов и программ, предназначенная для сбора и анализа данных о работоспособности IT-оборудования. Сервис собирает данные о состоянии серверов и систем, оповещая о возникших проблемах. Мониторинг осуществляется с помощью языка PromQL, а сама программа написана на Go и Ruby.

cProfile — это встроенный в Python модуль, который позволяет собирать статистику о частоте и продолжительности вызовов различных функций в коде.

Py-Spy — это профилировщик для программ на Python. Он позволяет отследить, в каком месте кода приложение проводит больше всего времени.

Интеграция приложений Flask с Docker

Чтобы упростить развертывание, гарантировать согласованность в разных средах и ускорить разработку, интегрируйте платформу контейнеризации Docker с вашим приложением Flask. В контейнере Docker содержатся все исполняемые файлы программы, включая ее код, среду выполнения, системные инструменты и библиотеки. Это означает, что в контейнерах Docker может храниться любая программа, которую можно установить на сервер. Благодаря такому подходу вы можете быть уверены, что программное обеспечение будет стабильно работать в любых условиях.

Установка Docker

Прежде чем приступить к контейнеризации приложения Flask, вам необходимо установить платформу для контейнеризации Docker на свою машину. Вы можете установить Docker с помощью менеджера пакетов. Выполните следующие команды:

```
sudo apt update
sudo apt install docker.io
```

Запуск и автоматизация работы службы Docker:

```
sudo systemctl start docker
sudo systemctl enable docker
```

После установки Docker следующим шагом будет создание контейнера для приложения Flask. Для этого необходимо создать `Dockerfile`, который представляет собой текстовый документ, содержащий все команды, которые пользователь может вызвать в командной строке для сборки образа.

Создание файла Dockerfile

Перейдите в корневой каталог вашего приложения Flask и создайте файл с именем `Dockerfile`:

```
# Используйте официальную среду выполнения Python
# в качестве родительского образа
FROM python:3.8-slim
# Установите рабочий каталог в контейнер
WORKDIR /app
# Скопируйте содержимое текущего каталога в контейнер по адресу /app
COPY . /app
# Установите все необходимые пакеты, указанные в файле
                                                    requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
# Сделайте порт 80 доступным для мира за пределами этого контейнера
EXPOSE 80
# Определите переменную окружения
ENV NAME World
# Запускаем app.py при запуске контейнера
CMD ["python", "app.py"]
```

Данный `Dockerfile` начинает работу с создания образа Python 3.8. `WORKDIR` определяет рабочую директорию внутри контейнера. Команда `COPY` получает исходный код приложения и копирует его в контейнер. Зависимости устанавливаются с помощью утилиты `pip` . Команда `EXPOSE` открывает доступ к порту 80 за пределами контейнера. `CMD` указывает, какую команду нужно запустить внутри контейнера.

Создание файла .dockerignore

Как и файл **.gitignore**, файл **.dockerignore** предотвращает копирование ваших локальных модулей и журналов отладки в образ Docker и возможную перезапись модулей, установленных в образе.

```
__pycache__/
*.pyc
*.pyo
*.pyd
.DS_Store
.git
.gitignore
venv/
```

Сборка образа Docker

Из каталога, в котором находится файл Dockerfile, выполните приведенную ниже команду для создания образа Docker. Флаг `-t` позволяет пометить образ, чтобы впоследствии его было легче найти с помощью команды `docker images`:

```
docker build -t yourusername/yourappname
```

Запуск контейнера Docker

После сборки образа запустите его в контейнере Docker:

```
docker run -p 4000:80 yourusername/yourappname
```

где `yourusername` — имя пользователя, `yourappname` — название вашего приложения.

Эта команда привязывает порт 4000 на вашей локальной машине к порту 80 в контейнере Docker, что позволит вам получить доступ к приложению Flask через localhost:4000 в вашем браузере.

Тестирование контейнера Docker

Чтобы убедиться, что ваш контейнер Docker работает правильно, перейдите в веб-браузере по адресу **http://localhost:4000**. Здесь

вы должны увидеть, что ваше приложение Flask работает в соответствии с настройками.

Благодаря использованию контейнеров Docker приложения на основе Flask можно легко разворачивать и поддерживать согласованность в средах разработки, тестирования и производства. Таким образом, устраняется проблема «на моей машине все работает». Среда Flask и ее зависимости инкапсулируются в Docker, что позволяет легко развертывать и управлять приложением независимо от базовой операционной системы или платформы.

Резюме

В этой главе мы подробно рассмотрели более сложные аспекты разработки и администрирования приложений, созданных на основе фреймворка Flask, расширив основные знания о Flask, которые мы получили в предыдущих главах. Чтобы улучшить управляемость и масштабируемость кодовой базы и для структурирования больших приложений, было рассмотрено использование расширения Flask Blueprints. Наши возможности по эффективному управлению различными установками и настройками были усовершенствованы, когда мы рассмотрели реализацию парадигмы Factory (Фабрика).

Далее мы занялись изучением Flask-RESTful, фреймворка для создания RESTful-сервисов, который упрощает процесс разработки RESTful веб-интерфейсов. Среди задач было непосредственное управление HTTP-методами и организация ответов в соответствии с ресурсами. Затем мы перешли к аутентификации и авторизации пользователей, целью которых было управление контролем доступа и создание способов подтверждения личности пользователя. Аутентификация и авторизация позволяют гарантировать, что только авторизованные пользователи смогут получить доступ к ресурсам.

Также были затронуты вопросы обработки ошибок и протоколирования. Эти моменты важны для поддержания надежности и работоспособности программы. Были рассмотрены способы эффективного управления и получения отчетности о сбоях в работе

приложения. Затем мы перешли к стратегиям оптимизации производительности, которые позволяют сделать наши приложения на основе Flask более динамичными и эффективными. Для достижения этих целей мы оптимизировали взаимодействие с базой данных и обработку запросов. Эти два компонента имеют решающее значение для сокращения времени загрузки и эффективного управления большим количеством пользовательских запросов.

И в конце раздела мы познакомились с технологией контейнеризации Docker для приложений на основе Flask, которая упрощает развертывание и гарантирует согласованность в различных настройках. Наши приложения Flask использовались для создания образов Docker, которые затем запускались в качестве контейнеров. Разработка профессиональных и высокопроизводительных веб-приложений с помощью Flask обеспечивает эффективность этих приложений, и эта глава посвящена передовым стратегиям и шаблонам, которые должны помочь в решении этой задачи. Разработчики, стремящиеся улучшить Flask-приложения и эффективнее справляться с большими проектами, могут счесть эти знания необходимыми.

ГЛАВА 4
Введение в FastAPI

Введение

В этой главе мы рассмотрим современный веб-фреймворк FastAPI, предназначенный для языка программирования Python и использующий обычные запросы для разработки API на языке Python 3.6+. Это быстрый, высокопроизводительный фреймворк. Независимо от того, используете ли вы Flask или другой фреймворк для Python, эта глава поможет вам освоить FastAPI и оценить его обширные возможности и преимущества, которые он обеспечивает для современной веб-разработки, особенно когда речь идет о создании RESTful API.

Для начала речь пойдет о переходе на FastAPI. Вы узнаете, чем отличается фреймворк FastAPI от фреймворка Flask и как эти изменения могут помочь в повышении качества вашего процесса разработки. Здесь можно вспомнить о двух главных отличиях фреймворка FastAPI от Flask. Это поддержка асинхронного программирования и аннотаций типов — современные возможности Python, повышающие производительность и продуктивность разработчиков.

Далее мы расскажем о том, как с помощью FastAPI создавать интерфейс RESTful API, продемонстрируем, как, используя функции Python и применяя модели библиотеки Pydantic, упростить определение конечных точек, сократить количество шаблонного кода и воспользоваться автоматической проверкой данных. Благодаря этим возможностям создавать API, которые соответствуют архитектуре REST, становится гораздо проще.

Далее мы рассмотрим простой и мощный механизм внедрения зависимостей FastAPI. С его помощью вы можете легко разде-

лить общую функциональность в приложении, например соединения с базой данных и схемы безопасности.

Для выполнения асинхронных операций с базами данных в *разделе «Расширенная интеграция баз данных с SQLAlchemy»* мы рассмотрим, как внедрить в фреймворк FastAPI программную библиотеку SQLAlchemy. Из этого же раздела мы также узнаем, как применить технологию ORM, чтобы воспользоваться асинхронными возможностями FastAPI для взаимодействия с базами данных с максимальной эффективностью.

Чтобы отправить электронную почту или обработать данные без задержки ответов клиентов, воспользуйтесь механизмом фоновых задач, реализованным в FastAPI, как показано в соответствующем разделе этой книги. В последнем *разделе «Интеграция FastAPI и Docker»* рассказывается о том, как создавать контейнеры для приложений, созданных на основе фреймворка FastAPI, чтобы упростить их развертывание и обеспечить согласованность сред в производстве, тестировании и разработке.

В этой главе вы подробно ознакомитесь со всеми возможностями FastAPI для создания безопасных, высокопроизводительных и быстрых веб-приложений. В конце книги читатели получат всю необходимую информацию, чтобы начать использовать FastAPI для создания собственных проектов или задуматься о переносе своих текущих приложений.

Переход на FastAPI

Прежде чем перейти с Flask или другого веб-фреймворка на FastAPI, необходимо ознакомиться с его основными особенностями и преимуществами, которые он предоставляет. FastAPI — это современный веб-фреймворк, использующий новейшие возможности Python для быстрой и удобной разработки API.

Основные различия между FastAPI и Flask

Производительность

FastAPI поддерживает асинхронный режим. FastAPI разработан на основе фреймворка Starlette для веб-частей и на основе биб-

лиотеки Pydantic для данных. FastAPI спроектирован как асинхронный фреймворк, и для поддержки асинхронной обработки запросов изначально может работать с ASGI-сервером Uvicorn. Благодаря этому он изначально быстрее, чем Flask, являющийся WSGI-фреймворком, в основе которого лежит синхронный подход.

Примечание переводчика

Starlette — ASGI-фреймворк, который используется при веб-разработке на Python. Он подходит для создания высокопроизводительных Asyncio-сервисов и может применяться как в роли самостоятельного фреймворка, так и в качестве ASGI-инструментария.

Некоторые особенности фреймворка:

- поставляется вместе с широким ассортиментом инструментов;
- поддерживает текстовый клиент, который построен на Requests Python;
- поддерживает WebSockets;
- поддерживает GraphQL — новый подход к разработке клиент-серверных взаимодействий;
- поддерживает внутрипроцессовые фоновые задачи;
- имеет широкий набор миддлеверов, предназначенных для работы с аутентификацией и авторизацией, CORS.

Asyncio — это модуль в стандартной библиотеке Python, который предоставляет инфраструктуру для написания одновременного кода с использованием асинхронных операций ввода-вывода.

Uvicorn — это легкий, супербыстрый ASGI-сервер, написанный на Python.

Аннотации типов данных и автоматическая валидация данных

FastAPI для объявления переменных в основном использует аннотации типов Python. Эта особенность нужна не только для наглядности разработки, она является неотъемлемой частью работы FastAPI. Фреймворк использует данные аннотации типов для автоматической проверки и сериализации данных.

```
from fastapi import FastAPI
from pydantic import BaseModel
```

```
app = FastAPI()
class Item(BaseModel):
name: str
description: str = None
price: float
tax: float = None
@app.post("/items/")
async def create_item(item: Item):
return {"name": item.name, "price": item.price}
```

В приведенном выше примере программы для автоматического преобразования данных из запросов в типы Python, проверки данных и генерации документации фреймворк FastAPI использует модели Pydantic (класс `Item`).

Внедрение зависимостей

FastAPI предоставляет простую в использовании, но чрезвычайно мощную систему внедрения зависимостей. Эта система позволяет создавать многократно используемые зависимости, которые можно внедрять в функцию маршрутизации.

> ***Примечание переводчика***
>
> Внедрение зависимостей — это передача извне зависимостей в класс. Внедрять зависимости можно через конструктор или отдельный метод-сеттер (сеттером его называют, потому что название этого метода начинается с set — «установить»). Функция получает нужные ей значения через аргументы, а класс — через конструктор.

```
from fastapi import Depends, FastAPI
def get_db():
db = DBSession()
try:
yield db
finally:
db.close()
@app.get("/items/")
async def read_items(db = Depends(get_db)):
items = db.get_items()
return items
```

В приведенном выше фрагменте кода `Depends` используется для создания зависимости, которая создает сессию базы данных для каждого запроса, гарантируя, что сессия будет закрыта после завершения запроса.

Встроенная интерактивная документация по API

FastAPI автоматически генерирует документацию по API в интерактивном режиме, используя инструменты Swagger UI и ReDoc. Документация включает в себя функции «Попробовать» (Try it out) и генерируется на основе вашего кода со всеми описаниями API, типами параметров и прочим.

Современные возможности Python

Фреймворк FastAPI предназначен для работы с современными версиями и возможностями Python. Фреймворк изначально поддерживает возможность асинхронного программирования, что идеально подходит для задач, связанных с вводом-выводом, которые часто встречаются в системах веб-интерфейсов.

Поняв суть этих фундаментальных различий, вы сможете использовать преимущества производительности и современные возможности Python для повышения эффективности ваших веб-приложений.

Создание RESTful API с помощью FastAPI

Встроенные функции фреймворка FastAPI предназначены для создания эффективных программных интерфейсов (API) для обмена данными между веб-сервисами и программами, а также управления ими. Применяя современные возможности Python и встроенный функционал FastAPI, можно в полной мере использовать возможности интерфейса RESTful API. В этом разделе мы рассмотрим, как эффективно использовать фреймворк FastAPI для создания интерфейса RESTful API, и покажем, как этот фреймворк позволяет улучшить и упростить процесс разработки по сравнению с традиционным способом написания кода.

Примечание переводчика

Еще одно определение RESTful API: это интерфейс, используемый двумя компьютерными системами для безопасного обмена информацией через Интернет.

Определение конечных точек RESTful

Фреймворк FastAPI обеспечивает простое определение конечных точек интерфейса RESTful. Здесь каждой конечной точке соответствует определенный HTTP-метод и путь. Для привязки функций Python к этим путям и методам используются декораторы. Ниже показано, как с помощью FastAPI определить простой API с использованием методов GET и POST:

```
from fastapi import FastAPI
from pydantic import BaseModel
app = FastAPI()
class Item(BaseModel):
name: str
description: str = None
price: float
tax: float = None
@app.get("/")
async def read_root():
return {"Hello": "World"}
@app.get("/items/{item_id}")
async def read_item(item_id: int):
return {"item_id": item_id}
@app.post("/items/")
async def create_item(item: Item):
return {"name": item.name, "price": item.price}
```

В приведенном выше примере программы:

- `@app.get("/")` и `@app.get("/items/{item_id}")` — это конечные точки для чтения данных.
- `@app.post("/items/")` — это конечная точка для создания данных, которая используется моделью библиотеки Pydantic (`Item`) для автоматической проверки и сериализации данных.

Примечание переводчика

Конечная точка — это отдельный URL-адрес, предоставляющий доступ API (интерфейсу прикладного программирования) к определенному ресурсу. Конечные точки при работе с API выполняют определенные задачи. Например, запрос данных или запуск процесса. Конечную точку API можно рассматривать как место, где две системы взаимодействуют друг с другом во время контакта. Например, взаимодействие API и сервера. API для выполнения своей задачи через конечную точку от сервера может получать необходимые ресурсы. Каждая конечная точка служит точкой доступа к ресурсам, необходимым API для работы.

Параметры пути и строки запросов

FastAPI предоставляет простой метод для определения параметров пути и строк запросов, необходимых для создания динамических конечных точек.

Параметры пути

Параметры пути могут определяться как аргументы функции. FastAPI автоматически интерпретирует аннотации типов для проверки и преобразования входных данных.

```
@app.get("/users/{user_id}")
async def read_user(user_id: int):
return {"user_id": user_id}
```

Параметры запросов

Параметры запроса могут быть определены как необязательные аргументы функции. Если задано значение по умолчанию, параметр является необязательным, в противном случае — обязательным.

```
@app.get("/items/")
async def read_items(q: str = None):
query = {"q": q}
return query
```

Использование тела запроса

Для более сложной структуры данных фреймворк FastAPI может работать с телами запросов в формате JSON, используя модели библиотеки Pydantic, которые обеспечивают проверку и структурирование данных.

```
@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Item):
return {"item_id": item_id, "name": item.name, "price": item.price}
```

В приведенном выше фрагменте кода `item` — это экземпляр модели Pydantic, который был извлечен из тела запроса, проверен и сериализован напрямую.

Обработка ответа

FastAPI предусматривает усовершенствованные механизмы обработки ответов, позволяющие осуществлять более точный контроль над HTTP-ответами.

Пользовательские коды состояния

Вы можете указать код состояния HTTP непосредственно в декораторе маршрута или в операторе возврата.

```
@app.post("/items/", status_code=201)
async def create_item(item: Item):
return item
```

Заголовки ответа

Для изменения заголовков ответа верните объект `Response`.

```
from fastapi import Response
@app.get("/items/{item_id}")
async def read_item(item_id: int):
return Response(content=f"Item ID: {item_id}",
media_type="text/plain")
```

Обработка ошибок

Для обработки ошибок в интерфейсе RESTful API фреймворк FastAPI обеспечивает создание HTTP-исключений, в которых находятся пользовательские заголовки и содержимое.

```
from fastapi import HTTPException
@app.get("/items/{item_id}")
async def read_item(item_id: int):
if item_id not in item_db:
raise HTTPException(status_code=404, detail="Item not found")
return {"item": item_db[item_id]}
```

В данном случае, если элемент не найден, это классифицируется как ошибка HTTP 404 и об этой ошибке выдается сообщение.

Существенную помощь в процессе быстрой разработки и поддержки API оказывает интуитивно понятное использование декораторов и автоматическое документирование. Благодаря этому фреймворк FastAPI становится хорошим выбором для программистов, которые хотят создавать скоростные и высокопроизводительные RESTful-сервисы.

Внедрение зависимостей

Один из шаблонов проектирования, который позволяет реализовать инверсию контроля (IoC), — это внедрение зависимостей (Dependency Injection). Этот шаблон облегчает процесс тестирования, позволяет управлять системой и способствует модульному построению приложения. Чтобы обеспечить бесперебойную работу обработчиков маршрутов и других компонентов приложения, разработчики могут использовать внедрение зависимостей с помощью фреймворка FastAPI, что позволит определять и управлять зависимостями, которые важны для работы приложения.

> ***Примечание переводчика***
>
> *Инверсия контроля* (инверсия управления) — это принцип в разработке программного обеспечения, при котором управление объектами или частями программы передается контейнеру или фреймворку.

Чаще всего этот принцип используется в контексте объектно-ориентированного программирования.

Преимущества этой архитектуры:

- отделение выполнения задачи от ее реализации;
- легкое переключение между различными реализациями;
- программа может быть разделена на модули;
- упрощенное тестирование программы путем изоляции компонента или проверки его зависимостей и обеспечения взаимодействия компонентов через контракты.

Инверсия управления может быть достигнута с помощью различных механизмов, таких как:

- шаблон проектирования «Стратегия»;
- шаблон «Локатор служб»;
- шаблон «Фабрика»;
- внедрение зависимостей (DI).

Описание внедрения зависимостей в FastAPI

Технология внедрения зависимостей с помощью фреймворка FastAPI подразумевает объявление конкретных зависимостей непосредственно в параметрах функции пути. Затем эти зависимости обрабатываются фреймворком FastAPI в процессе выполнения запроса. Фреймворк использует аннотации типов Python для определения того, что нужно внедрить.

Примечание переводчика

Функция пути определяет последовательность выполнения операций в программе. Эта функция указывает очередность и порядок выполнения действий. Функция пути задает логику выполнения программы и определяет, как программа будет реагировать на различные условия и входные данные. Функция пути позволяет программисту контролировать поведение программы и принимать решения на основе внешних факторов.

Как работает внедрение зависимостей в FastAPI?

Определение зависимостей

Зависимости в фреймворке FastAPI обычно определяются как функции. Эти функции могут выполнять различные задачи. На-

пример, установка соединений с базой данных, аутентификация пользователей или любые другие повторяющиеся действия в различных частях приложения.

```
from fastapi import Depends, HTTPException
def get_db():
try:
db = Database.connect()
yield db
finally:
db.disconnect()
def get_current_user(db=Depends(get_db)):
user_id = db.get_current_user_id()
if not user_id:
raise HTTPException(status_code=404, detail="User not found")
return user_id
```

В приведенном выше фрагменте кода `get_db` — это зависимость, которая создает соединение с базой данных, а `get_current_user` — это зависимость, которая использует другую зависимость (`get_db`) для получения идентификатора текущего пользователя из базы данных.

Использование зависимостей в обработчиках маршрутов

Как только зависимости определены, их можно внедрять в процесс определения пути, объявляя с помощью класса `Depends` путь в качестве параметра по умолчанию.

```
from fastapi import FastAPI, Depends
app = FastAPI()
@app.get("/users/me")
def read_current_user(user_id: int = Depends(get_current_user)):
return {"user_id": user_id}
```

В приведенном выше примере программы `read_current_user` — это обработчик маршрута, который зависит от функции `get_current_user`. Фреймворк FastAPI выполняет эти зависимости в фоновом режиме и вставляет результат (`user_id`) в функцию работы с маршрутом.

Работа с зависимостями в больших приложениях

При работе с большими приложениями в FastAPI может появиться множество зависимостей, которые необходимо внедрить в разных частях приложения. FastAPI позволяет структурировать эти зависимости в иерархическом порядке, где высокоуровневые зависимости могут зависеть от низкоуровневых.

```
def get_api_key(db=Depends(get_db)):
key = db.get_api_key()
if not key:
raise HTTPException(status_code=403, detail="API key invalid")
return key
@app.get("/data")
def read_data(api_key: str = Depends(get_api_key)):
return {"data": "secret data"}
```

В этом случае `read_data` зависит от `get_api_key`, который, в свою очередь, зависит от `get_db`. Эта иерархия автоматически структурируется фреймворком FastAPI, гарантируя, что каждая функция получает те зависимости, которые ей необходимы для эффективной работы. Встроенная в FastAPI поддержка внедрения зависимостей с помощью простых функций Python и класса `Depends` упрощает то, что в противном случае могло бы стать сложной и громоздкой частью разработки приложений.

Расширенная интеграция баз данных с SQLAlchemy

Технология объектно-реляционного отображения (ORM) позволяет эффективно манипулировать данными и извлекать их, а приложения на основе фреймворка FastAPI благодаря такой интеграции получают доступ к расширенным возможностям библиотеки SQLAlchemy по управлению базами данных. Данная интеграция позволяет легко использовать Python-код для взаимодействия с различными базами данных и использовать возможности SQLAlchemy для обработки расширенных запросов, транзакций и многого другого.

Настройка SQLAlchemy с помощью FastAPI

Установка необходимых пакетов

Прежде всего убедитесь, что у вас установлены пакеты SQLAlchemy и базы данных, поддерживающие асинхронные операции:

```
pip install sqlalchemy databases[sqlite]
pip install asyncpg
```

Это для PostgreSQL, для других баз данных установите соответствующий драйвер.

Настройка URL базы данных

Введите строку подключения к базе данных, которую библиотека SQLAlchemy будет использовать для подключения к вашей базе данных. Это можно указать в настройках приложения или непосредственно в коде.

```
DATABASE_URL = "sqlite:///./test.db "
# Для PostgreSQL: postgresql://user:password@localhost/dbname
```

Создание базы данных и таблиц

Используя технологию объектно-реляционного отображения и библиотеку SQLAlchemy, определите модели, которые библиотека SQLAlchemy будет использовать для создания таблиц в вашей базе данных. Ниже приведен пример настройки модели:

```
from sqlalchemy import create_engine, Column, Integer, String,
                                                        MetaData
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(SQLALCHEMY_DATABASE_URL,
connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)
Base = declarative_base()
class User(Base):
__tablename__ = "users"
```

```
id = Column(Integer, primary_key=True, index=True)
username = Column(String, unique=True, index=True)
email = Column(String, index=True)
full_name = Column(String)
disabled = Column(Boolean, default=False)
# Создание таблиц базы данных
Base.metadata.create_all(bind=engine)
```

Этот код устанавливает базу данных `SQLite`, определяет модель `User` и создает соответствующую таблицу в базе данных.

Интеграция SQLAlchemy с FastAPI

Для интеграции библиотеки SQLAlchemy в приложение, созданное с помощью фреймворка FastAPI, обычно используется внедрение зависимостей, которые обеспечивают ограничение сессии для каждого запроса.

Зависимость от сеанса работы с базой данных

Создайте зависимость, которую обработчики маршрутов смогут использовать для получения сессии и выполнения операций с базой данных.

```
from fastapi import Depends, FastAPI, HTTPException
from sqlalchemy.orm import Session
app = FastAPI()
# Зависимость
def get_db():
db = SessionLocal()
try:
yield db
finally:
db.close()
```

Выполнение операций с базой данных

Используйте сессию, предоставленную зависимостью, для выполнения операций CRUD (Create — создание, Read — чтение, Update — обновление, Delete — удаление). Ниже описано, как можно получить данные из базы данных:

```
@app.get("/users/{user_id}", response_model=UserSchema)
async def read_user(user_id: int, db: Session = Depends(get_db)):
db_user = db.query(User).filter(User.id == user_id).first()
if db_user is None:
raise HTTPException(status_code=404, detail="Пользователь не найден")
return db_user
```

Асинхронная обработка

Для реализации настоящего выполнения асинхронных операций при работе с базами данных в фреймворке FastAPI совместно с ядром SQLAlchemy (не ORM) необходимо использовать пакет databases. Такой подход позволяет выполнять запросы к базе данных асинхронно.

Настройка асинхронного подключения к базе данных

```
from databases import Database
database = Database(DATABASE_URL)
```

Подключение и отключение событий

```
@app.on_event("startup")
async def startup():
await database.connect()
@app.on_event("shutdown")
async def shutdown():
await database.disconnect()
```

Использование асинхронных запросов

Запросы к базе данных выполняются в асинхронном режиме с использованием объекта базы данных.

```
@app.get("/items/")
async def read_items():
query = items.select()
return await database.fetch_all(query)
```

Более сложные, эффективные и высокопроизводительные приложения станут реальностью, если использовать все возможности библиотеки SQLAlchemy и технологии ORM при маршрутизации

и обработке объектов, а также асинхронное выполнение запросов в ядре библиотеки SQLAlchemy.

Реализация фоновых задач

Основы фоновых задач в FastAPI

Одним из эффективных методов работы с операциями, которые занимают много времени или не являются критичными для основного ответа на запрос, является их реализация в фоновом режиме работы приложения, созданного с помощью фреймворка FastAPI. В качестве примера можно привести операции по отправке электронной почты, обработке файлов или обновление баз данных, когда пользователю не стоит терять время, ожидая ответа.

FastAPI предоставляет простой и интуитивно понятный способ определения фоновых задач, которые выполняются после отправки ответа клиенту. Эта функциональность встроена непосредственно в FastAPI и использует стандартные возможности Python, что делает реализацию простой и не требующей использования внешних библиотек.

Как определить фоновые задачи?

FastAPI содержит класс `BackgroundTasks`, который можно использовать для добавления фоновых задач, выполняемых после завершения запроса.

Ниже описано, как использовать этот класс:

```
from fastapi import FastAPI, BackgroundTasks
app = FastAPI()
def write_log(message: str):
with open("log.txt", "a") as log:
log.write(f"{message}\n")
@app.post("/send-notification/")
async def send_notification(email: str, background_tasks:
BackgroundTasks):
```

```
background_tasks.add_task(write_log, message=f"notification sent to
{email}")
return {"message": "Notification sent in the background"}
```

В приведенном выше примере программы, когда вызывается конечная точка `/sendnotification/`, планируется запуск функции `write_log` в фоновом режиме. API немедленно отправляет сообщение пользователю. А сама операция записи в журнал выполняется в фоновом режиме, не вынуждая пользователя тратить время на ожидание.

Реализация более сложных фоновых операций

Для более сложных или ресурсоемких фоновых задач вам может понадобиться интеграция более эффективного решения, такого как Celery, с брокером сообщений, например RabbitMQ или Redis. Далее приведена базовая схема того, как это можно реализовать.

Примечание переводчика

Celery — это система очередей задач, которая позволяет эффективно управлять асинхронными задачами в Python. Данная система позволяет разделять сложные процессы на более мелкие задачи, которые могут выполняться параллельно и асинхронно, что увеличивает производительность и масштабируемость приложения.

Основные компоненты Celery:

- Task — базовый элемент Celery, который представляет собой отдельную задачу, выполняемую асинхронно;
- Queue — очередь задач, которые будут выполнены асинхронно;
- Worker — процесс, который запускает и обрабатывает задачи из очереди;
- Broker — посредник между задачами и worker'ами. Он предоставляет очередь задач и хранит информацию о задачах, которые должны быть выполнены.

Настройка Celery

Установите Celery и выберите брокер сообщений (RabbitMQ, Redis и т. д.), затем настройте Celery в вашем приложении, созданном с помощью фреймворка FastAPI.

```
from celery import Celery
def make_celery(app):
celery = Celery(
app.import_name,
backend=app.config['CELERY_RESULT_BACKEND'],
broker=app.config['CELERY_BROKER_URL']
)
celery.conf.update(app.config)
class ContextTask(celery.Task):
def __call__(self, *args, **kwargs):
with app.app_context():
return self.run(*args, **kwargs)
celery.Task = ContextTask
return celery
celery = make_celery(app)
```

Определение задач Celery

Вы можете определить задачи, которые Celery будет выполнять асинхронно. Ниже описано, как определить простую задачу Celery.

```
@celery.task()
def send_email(email: str):
# логика отправки письма
pass
```

Запуск задач Celery из FastAPI

Теперь вы можете по мере необходимости запускать данные задачи Celery через маршруты FastAPI.

```
@app.post("/send-email/")
async def send_email_endpoint(email: str):
send_email.delay(email=email)
return {"message": "Email отправляется в фоновом режиме"}
```

Внутренние фоновые задачи, встроенные в FastAPI, легко использовать, и их более чем достаточно для выполнения всех базовых задач. В более сложных ситуациях для задач, требующих жесткого управления, масштабируемости или распределенной обработки, лучше использовать очередь задач, например Celery.

Интеграция FastAPI и Docker

Одним из способов использования Docker с FastAPI является контейнеризация вашего приложения, созданного с помощью FastAPI. В этом случае вам будет проще тестировать, разрабатывать и запускать приложение хотя и в разных, но совместимых средах. В ходе процесса контейнеризации приложение на основе FastAPI и все его зависимости упаковываются в контейнер Docker. Затем контейнер можно легко развернуть и управлять приложением.

Установка Docker

Перед тем как приступить к выполнению контейнерной обработки приложения, созданного на основе FastAPI, убедитесь, применив показанный ниже способ, что в вашей среде разработки Docker установлен.

```
sudo apt update
sudo apt install docker.io
sudo systemctl start docker
sudo systemctl enable docker
```

После установки добавьте в группу Docker нового пользователя, чтобы выполнять команды Docker без утилиты sudo.

```
sudo usermod -aG docker ${USER}
su - ${USER}
```

> ***Примечание переводчика***
>
> Sudo — это утилита для операционных систем семейства Linux, позволяющая пользователю запускать программы с привилегиями другой учетной записи, как правило, суперпользователя.

Создание Docker-файла для FastAPI

Чтобы заключить приложение, созданное на основе фреймворка FastAPI в контейнер, в корневом каталоге вашего проекта FastAPI создайте Docker-файл. Этот файл определяет шаги по созданию образа Docker, что показано ниже.

```
# Используйте официальную среду выполнения Python в качестве
# базового образа
FROM python:3.8
# Установите рабочий каталог в контейнере
WORKDIR /code
# Скопируйте содержимое текущего каталога в контейнер по адресу /code
COPY ./ /code
# Установите все необходимые пакеты, указанные
# в файле requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
# Сделайте порт 8000 доступным для окружающей среды
# за пределами этого контейнера
EXPOSE 8000
# Определите переменную окружения
ENV NAME World
# Запустите приложение
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Ниже показано, как работает приведенный выше сценарий:

1. Создание Docker-файла начинается с запуска образа Python 3.8.
2. Создает рабочий каталог **/code** внутри контейнера.
3. Копирует приложение FastAPI в контейнер Docker.
4. Устанавливает зависимости из файла **requirements.txt**.
5. Открывает порт 8000 для сервера Uvicorn.
6. Запускает с помощью Uvicorn приложение на основе FastAPI.

Создание образа Docker

Перейдите в каталог, в котором находится ваш Docker-файл, и для создания образа Docker выполните следующую команду:

```
docker build -t fastapi-app .
```

Эта команда создает образ Docker, используя Docker-файл, находящийся в текущем каталоге, и помечает образ как `fastapi-app`.

Запуск приложения FastAPI в контейнере Docker

После создания образа запустите ваше приложение, созданное с помощью фреймворка FastAPI в контейнере Docker, выполнив команду:

```
docker run -d -p 8000:8000 fastapi-app
```

Флаг `-d` запускает контейнер в режиме автономной работы, позволяя ему работать в фоновом режиме. А флаг `-p 8000:8000` сопоставляет порт 8000 контейнера с портом 8000 на хосте, что позволяет получить доступ к приложению на основе FastAPI через `localhost:8000`.

Проверка работоспособности приложения

Чтобы убедиться, что ваше приложение FastAPI корректно работает в Docker, перейдите в веб-браузере по адресу **http://localhost:8000**. Вы должны увидеть, что ваше приложение, созданное с помощью фреймворка FastAPI, отвечает на все ваши запросы в соответствии с настройками программы.

Выполнив эти шаги, вы сможете запустить свое приложение, созданное с помощью фреймворка FastAPI в контейнере Docker. Кроме того, вам удалось настроить среду Docker. И при такой конфигурации ваше приложение всегда будет работать в «песочнице», независимо от любой другой среды, будь то локальная машина, тестовая среда или рабочий сервер.

Резюме

В этой главе мы подробно познакомились с FastAPI — современным веб-фреймворком, предназначенным для создания высокопроизводительных API на языке Python. Сначала мы узнали, почему FastAPI пришел на смену Flask. В фреймворке FastAPI основное внимание было уделено поддержке асинхронной работы, а также возможностям автоматического подтверждения данных с помощью аннотаций типов и библиотеки Pydantic. Получив такой фундамент, мы с помощью веб-фреймворка FastAPI смогли более глубоко погрузиться в детали разработки интерфейса

RESTful API. После этого мы узнали, как FastAPI упрощает разработку RESTful-сервисов. Благодаря отсутствию необходимости в повторяющемся коде веб-фреймворк упрощает такие привычные процессы, как обработка запросов, проверка данных и сериализация ответов. Благодаря такому упрощенному методу становится выше производительность и надежность приложений, одновременно повышается производительность разработчиков.

Еще одна важная тема, которую мы рассмотрели, — внедрение зависимостей. Благодаря обработке зависимостей многократно используемыми функциями передовая система внедрения зависимостей веб-фреймворка FastAPI делает код чище и проще в сопровождении. Подключения к базе данных и аутентификация пользователей — это примеры общих шаблонов, и данная система крайне необходима для эффективной работы с этими шаблонами. В данной главе также рассматривается расширенная интеграция базы данных SQLAlchemy и настройки FastAPI с акцентом на выполнение с базой данных асинхронных операций, чтобы максимально использовать преимущества неблокирующих функций FastAPI. Такая интеграция гарантирует быстрое и масштабируемое взаимодействие с базой данных.

Кроме того, здесь было рассказано, как запускать фоновые задачи в FastAPI, которые крайне важны для процессов, выполняющихся слишком долго. Такие задачи, как отправка электронной почты или обработка больших объемов данных, могут выполняться параллельно, не замедляя основной поток приложения, что обеспечивает непрерывную работу пользователей. Последним разделом главы стала демонстрация того, как интегрировать приложения, созданные на основе фреймворка FastAPI, с инструментом для ускоренной разработки приложений в контейнерах Docker. Здесь мы рассказали о том, как выполнить контейнеризацию приложения, созданного на основе FastAPI, т. е. упаковать его в Docker, чтобы упростить развертывание и масштабируемость в разных средах. Независимо от того, находится ли приложение в производстве, тестировании или разработке, Docker гарантирует его совместимость.

В итоге в этой главе читатели узнали, как использовать веб-фреймворк FastAPI для создания мощных, эффективных и простых в развертывании веб-приложений.

ГЛАВА 5
Работа с базами данных

Введение

Чтобы создавать надежные и расширяемые веб-приложения, нужно хорошо понимать механизмы управления базами данных и стратегий оптимизации. Изучение этих механизмов — тема этой главы. Здесь мы поговорим о современных методах взаимодействия с базами данных, охватывающих широкий спектр технологий и методологий: от классических реляционных баз данных до современных NoSQL-систем.

Сначала мы рассмотрим две широко распространенные реляционные базы данных — MySQL и PostgreSQL. В этом разделе вы узнаете подробные характеристики баз данных, их преимущества и рекомендации по применению каждой системы управления базами данных. Также здесь будет рассказано о плюсах и минусах каждого варианта и о том, как правильно выбрать для вашего приложения подходящий вариант.

Затем мы познакомимся с базами данных NoSQL, в частности с MongoDB, преимуществом которой является высокая степень масштабируемости, доступность и очень хорошая производительность. Вы узнаете, как работать с базой данных MongoDB, начиная с ориентированной на документы структуры, об основных отличиях от реляционных баз данных в моделировании и поиске информации, а также о ее преимуществах и недостатках.

Далее в главе рассматриваются основные принципы проектирования баз данных. Целостность информации, повышение производительности и простота обслуживания системы — все это присуще хорошо спроектированной базе данных. Для эффективного

и надежного проектирования структуры базы данных необходимы знания о нормализации, отношениях и ключевых ограничениях. Затем мы рассмотрим более сложные методы формирования запросов и операции CRUD (Create — создание, Read — чтение, Update — обновление и Delete — удаление). В этой части вы узнаете, как работать с более сложными структурами данных, как получать данные разными способами, как написать подробный запрос и как можно более эффективно управлять информацией в различных системах баз данных.

Затем мы расскажем о контроле версий и миграции баз данных. Чтобы обеспечить согласованность сред разработки, тестирования и производства, вы узнаете, как управлять и отслеживать происходящие со временем изменения в схеме базы данных. Системы объектно-реляционного отображения (ORM) в Python и интеграция баз данных — еще одна важная тема. В этом разделе рассказывается о том, как ORM упрощают взаимодействие с базой данных, позволяя работать с сущностями как с объектами Python. Это помогает повысить производительность труда разработчиков, избавляя их от написания сложных SQL-запросов.

В завершение главы мы рассмотрим некоторые стратегии кеширования, которые помогут оптимизировать работу баз данных. Снижая нагрузку на базу данных и уменьшая время задержки при получении информации, кеширование творит чудеса с точки зрения производительности приложений, работающих с базами данных. В этой главе вы узнаете все, что необходимо знать для создания высокопроизводительных веб-приложений, включая управление базами данных, эффективное манипулирование данными и оптимизацию взаимодействия с базами данных.

MySQL и PostgreSQL

Благодаря возможностям хранения, поиска и манипулирования данными реляционные базы данных являются основой многих современных веб-приложений. Среди реляционных баз данных наиболее широко известны PostgreSQL и MySQL. Несмотря на их сходство и широкую поддержку SQL-запросов, существует ряд различий между этими двумя версиями баз данных. Благодаря

этим различиям вы можете выбрать наиболее подходящий вариант.

MySQL

MySQL — это реляционная система управления базами данных с открытым исходным кодом, известная своей надежностью и простотой использования. Она особенно востребована в веб-приложениях и является частью стека LAMP (Linux, Apache, MySQL, PHP/Python/Perl). База данных MySQL считается высокоскоростной и предназначена для эффективной работы с приложениями, требующими большого объема чтения. По умолчанию здесь используется механизм хранения данных InnoDB, который обеспечивает полное соответствие стандартам ACID (Atomicity — атомарность, Consistency — согласованность, Isolation — изолированность, Durability — долговечность) и поддерживает транзакции, что очень важно для поддержания целостности данных.

MySQL поддерживает репликацию master-slave (ведущий — ведомый), позволяющую реплицировать данные с одного ведущего сервера базы данных MySQL (master) на один или несколько ведомых серверов базы данных MySQL (slaves). Репликация используется в основном для реализации стратегий масштабирования, резервного копирования данных и резервирования. Система MySQL известна своей простотой и удобством использования. Эта база данных может стать хорошим выбором для небольших и средних веб-проектов, которым нужна простая в настройке и управлении база данных.

PostgreSQL

PostgreSQL или просто Postgres — это система управления реляционными базами данных с открытым исходным кодом, в которой особое внимание уделяется возможности расширения и соответствию требованиям SQL. Эта база данных с точки зрения технологий и возможностей считается более совершенной, чем MySQL. Главное отличие PostgreSQL — способность обрабатывать сложные запросы и выполнять несколько процессов одно-

временно (конкурентность). Эта база данных поддерживает различные средства повышения производительности, такие как индексы с выражениями, частичные индексы и широкий набор типов индексов. PostgreSQL позволяет использовать расширенные типы данных и оптимизировать производительность, что обеспечивает работу в сложных средах с большим объемом данных. К таким данным относятся геопространственные данные, созданные с помощью PostGIS, пользовательские типы данных и сложные механизмы блокировки.

У PostgreSQL есть широкие возможности расширения. Например, вы можете определять собственные типы данных, создавать пользовательские функции и даже писать код на разных языках программирования без перекомпиляции базы данных. PostgreSQL известен своим высоким уровнем соответствия ACID. В нем поддерживаются сложные транзакции SQL и большое внимание уделяется соответствию стандарту SQL.

Выбор между MySQL и PostgreSQL

Принимая решение об использовании MySQL или PostgreSQL для разработки внутренних приложений на языке программирования Python, нужно учесть несколько моментов.

- Требования к приложениям.
  - Если вашему приложению необходимо полное соответствие требованиям ACID и сложная поддержка транзакций, лучше выбрать PostgreSQL.
  - Если ваше приложение ориентировано на чтение и не предполагает проведения сложных транзакционных операций или ему не нужны расширенные возможности, предоставляемые PostgreSQL, вам может подойти MySQL.
- Масштабируемость.
  - Обе базы данных обеспечивают хорошие возможности масштабирования, но выбор может зависеть от типа необходимого уровня масштабируемости. Для приложений, предполагающих интенсивное чтение данных, возможно-

сти репликации MySQL делают эту базу предпочтительным выбором.

- PostgreSQL обеспечивает лучшую масштабируемость при записи и больше подходит для приложений с большой нагрузкой одновременно выполняемых транзакций.

♦ Поддержка и совместимость.

- И MySQL, и PostgreSQL поддерживаются крупными сообществами и обладают обширной документацией. Сообщество PostgreSQL особенно сильно в сфере открытых исходных кодов. Здесь доступно множество инструментов и расширений от сторонних разработчиков.
- MySQL, принадлежащая корпорации Oracle, имеет коммерческую поддержку, что может стать решающим фактором для предприятий, нуждающихся в гарантированной поддержке.
- Подумайте, с какими базами данных необходимо взаимодействовать вашему приложению. Если вы уже используете инструменты, которые лучше интегрируются с MySQL или PostgreSQL, это может повлиять на ваш выбор.

И PostgreSQL, и MySQL — хороший выбор для поддержки бесперебойной работы внутренних компонентов приложений, написанных на языке Python. Но у этих баз данных немного разные требования. Приложениям, для работы которых необходимы сложные запросы, высокая целостность данных или специализированные функции, такие как поддержка географических данных, может больше подойти база данных PostgreSQL, поскольку в ней предусмотрены более сложные функции, которые доступны сразу после установки. При проектировании приложений с более простой конфигурацией репликации, быстрой настройкой и хорошей производительностью при больших нагрузках в режиме чтения, разработчикам стоит обратить внимание на базу данных MySQL. Решения должны приниматься с учетом уровня технической подготовки команды разработчиков, будущих требований к масштабируемости и специфики проекта.

MongoDB

Для эффективной работы многих современных приложений с гибкими информационными структурами, которым нужны масштабируемые базы данных, предлагается MongoDB, мощная NoSQL-база данных. Этот выбор обусловлен ее высокой доступностью, простотой масштабирования и высокой производительностью. В отличие от реляционных баз данных, которые опираются на таблицы и заранее определенную схему, MongoDB ориентирована на документы. Это позволяет создать более универсальную и адаптируемую модель данных, поскольку информация хранится в документах BSON (Binary JSON) с динамической схемой.

Описание MongoDB и модели документов

В базе данных MongoDB используется модель документов, которая представляет собой полуструктурированный формат данных. Эта модель невероятно гибкая и позволяет хранить данные без необходимости заранее определять их структуру. Каждый документ может иметь свою уникальную структуру с различными полями, а тип данных каждого поля может варьироваться от документа к документу.

Документы

Документы — это основные единицы данных в MongoDB, схожие со строками в реляционной базе данных, но гораздо более гибкие. Документ в MongoDB — это карта имен полей и значений. Эти значения могут содержать массивы и вложенные документы, что дает возможность хранить сложные иерархические структуры в одном документе.

Коллекции

Коллекции — это аналог таблиц для реляционных баз данных. Коллекции могут содержать несколько документов. В отличие от таблиц в реляционных базах данных, в коллекциях не применяются ограничения схемы, а значит, разные документы могут состоять из разных полей.

Преимущества MongoDB

- *Масштабируемость.* MongoDB разработана с учетом требований масштабируемости. Здесь поддерживается горизонтальное масштабирование за счет распределения данных по нескольким машинам.
- *Производительность.* MongoDB обеспечивает высокую производительность как при чтении, так и при записи. Ее механизм хранения оптимизирован для эффективного хранения и извлечения данных, что повышает общую производительность.
- *Гибкость.* Благодаря бессхемной организации MongoDB позволяет разрабатывать приложения без необходимости предварительного определения схемы или внесения в схему значительных изменений по мере возникновения новых требований.
- *Высокая доступность.* Средства репликации MongoDB, называемые наборами реплик, обеспечивают автоматическое восстановление после сбоев и избыточность данных, что гарантирует высокую доступность вашего приложения.

Ключевые особенности MongoDB

- MongoDB обеспечивает поиск по полю, поиск по диапазону и поиск по регулярным выражениям. Запросы могут возвращать определенные поля в документах и включать пользовательские функции JavaScript.
- Любое поле в документе MongoDB может быть проиндексировано. Индексы очень важны для повышения производительности поиска.
- В MongoDB реализована система агрегации, основанная на концепции конвейеров обработки данных. Документы поступают на многоступенчатый конвейер, который преобразует их в агрегированные результаты.
- Для хранения и получения больших файлов, таких как изображения, видео или большие массивы данных, MongoDB предоставляет GridFS — спецификацию для хранения и полу-

чения файлов, которые превышают ограничение BSON-documentsize в 16 Мбайт.

Интеграция MongoDB

Для работы MongoDB с приложениями, созданными на платформе Python, обычно используется библиотека `pymongo`, которая предоставляет соответствующие инструменты.

Установка

Установите `pymongo` с помощью менеджера пакетов `pip`.

```
pip install pymongo
```

Подключение к MongoDB

Создайте скрипт на языке Python, который установит соединение с экземпляром MongoDB.

```
from pymongo import MongoClient
# Подключитесь к серверу MongoDB, работающему на localhost
# по порту 27017
client = MongoClient('localhost', 27017)
# Доступ к базе данных с именем 'test_database'
db = client.test_database
```

Операции

- Создание и вставка документов.

```
# Доступ к коллекции с именем 'test_collection'
collection = db.test_collection
# Вставка документа
post = {"author": "John", "text": "Первый пост!"}
collection.insert_one(post)
```

- Запросы.

```
# Найдите один документ
import pprint
pprint.pprint(collection.find_one({"author": "John"}))
```

- Обновление.

```
# Обновление документа
collection.update_one({"author": "John"},
{"$set": {"text": "Обновленный пост"}})
```

- Удаление.

```
# Удалить документ
collection.delete_one({"author": "John"})
```

Адаптируемость к различным типам данных, широкие возможности по обработке запросов и масштабируемость делают эту базу данных хорошим выбором для современных приложений, которым требуется мощная и универсальная база данных.

Принципы проектирования баз данных

Разработка эффективных, масштабируемых и легко обслуживаемых баз данных в значительной степени зависит от принципов их проектирования. Если ваша база данных хорошо спроектирована, ваше приложение будет работать без сбоев и сможет поддерживать все изменения по мере их возникновения. Тип данных, предполагаемая нагрузка и типы запросов — все это важные моменты при выборе оптимальной схемы базы данных для внутреннего приложения.

Основные принципы проектирования баз данных

Нормализация

Нормализация — это процесс структурирования реляционной базы данных в соответствии с серией так называемых нормальных форм с целью сокращения избыточности данных и повышения их целостности. Нормализация предполагает разбиение одной таблицы на несколько таблиц с меньшей степенью резервирования, но без потери информации.

- *Первая нормальная форма (1NF).* Гарантирует, что каждый столбец таблицы является атомарным, а каждая строка содержит уникальные данные.

- *Вторая нормальная форма (2NF).* Требуется, чтобы база данных была в форме 1NF, а все столбцы, которые не зависят от первичного ключа, должны быть удалены.
- *Третья нормальная форма (3NF).* База данных находится в форме 3NF, если соответствует форме 2NF, а все столбцы в таблице зависят от первичного ключа, но независимы друг от друга.

Модель отношений между сущностями

Модель Entity-Relationship (ER) позволяет наглядно представить и спроектировать структуру базы данных. Модель включает в себя определение сущностей (объектов, о которых нужно хранить данные) и их отношений друг с другом.

Пример разработки схемы базы данных

Предположим, что вы разрабатываете систему для управления университетом, в которой необходимо управлять работой студентов, курсами и зачислением студентов на эти курсы. Ниже приведена схема базы данных.

Организация

- `Students` (студенты): содержит подробную информацию о студентах.
- `Courses` (курсы): содержит информацию о курсах.
- `Enrollments` (зачисления): представляет отношения между студентами и курсами.

Атрибуты

- `Students` (студенты):
  - `Student_ID` (первичный ключ);
  - `Name` (имя);
  - `Email` (электронная почта);
  - `Date_of_birth` (дата рождения).

- Courses (курсы):
  - Course_ID (первичный ключ);
  - Course_Name (название курса);
  - Course_Description (описание курса).
- Enrollments (зачисления):
  - Enrollment_ID (первичный ключ);
  - Student_ID (идентификатор студента — внешний ключ);
  - Course_ID (идентификатор курса — внешний ключ);
  - Enrollment_Date (дата зачисления).

Это отношение «многие-ко-многим» представлено в таблице Enrollments, в которой содержатся внешние ключи, ссылающиеся на таблицы Students и Courses, тем самым устанавливая связь между ними.

SQL-код для создания таблиц

```
CREATE TABLE Students (
Student_ID INT PRIMARY KEY,
Name VARCHAR(100),
Email VARCHAR(100),
Date_of_birth DATE
);
CREATE TABLE Courses (
Course_ID INT PRIMARY KEY,
Course_Name VARCHAR(100),
Course_Description TEXT
);
CREATE TABLE Enrollments (
Enrollment_ID INT PRIMARY KEY,
Student_ID INT,
Course_ID INT,
Enrollment_Date DATE,
FOREIGN KEY (Student_ID) REFERENCES Students(Student_ID),
FOREIGN KEY (Course_ID) REFERENCES Courses(Course_ID)
);
```

Схема нашей системы управления университетом построена таким образом, чтобы эффективно управлять данными о студентах, курсах и зачислениях. Внутренние операции будут выполняться эффективно и надежно благодаря уменьшению избыточности и обеспечению целостности базы данных.

Операции CRUD

Далее мы рассмотрим реализацию операций CRUD (Create — создание, Read — чтение, Update — обновление и Delete — удаление). Благодаря этим операциям в вашем приложении становится возможным эффективное управление данными. Для выполнения этих операций над данными, содержащимися в созданной нами схеме, мы рассмотрим методы работы с SQL.

Создание записей (CREATE)

Чтобы добавить новые данные в таблицу, мы используем оператор `INSERT` языка SQL. Ниже показано, как можно добавить записи в таблицы `Students`, `Courses` и `Enrollments`.

```
-- Вставка данных в таблицу Students
INSERT INTO Students (Student_ID, Name, Email, Date_of_birth)
VALUES (1, 'John Doe', 'john.doe@gitforgits.com', '2000-01-01');
-- Вставка данных в таблицу Courses
INSERT INTO Courses (Course_ID, Course_Name, Course_Description)
VALUES (101, 'Introduction to Psychology', 'A foundational course
covering basic principles of psychology.');
-- Вставка данных в таблицу Enrollments
INSERT INTO Enrollments (Enrollment_ID, Student_ID, Course_ID,
Enrollment_Date)
VALUES (1, 1, 101, '2021-09-01');
```

Эти операторы SQL добавляют студента, курс и запись о зачислении, связывающую студента с курсом.

Чтение записей (READ)

Чтение данных, или запрос к таблицам, выполняется с помощью оператора `SELECT` языка SQL. Ниже приведены примеры получения данных из таблиц.

```
-- Выберите все поля из раздела Студенты
SELECT * FROM Students;
-- Выберите конкретный курс по идентификатору
SELECT * FROM Courses WHERE Course_ID = 101;
-- Выберите все зачисления, включая сведения о студентах и курсах
SELECT s.Name, c.Course_Name, e.Enrollment_Date
FROM Enrollments e
JOIN Students s ON e.Student_ID = s.Student_ID
JOIN Courses c ON e.Course_ID = c.Course_ID;
```

С помощью этих запросов можно получить данные обо всех студентах, конкретных курсах и подробные записи о зачислении, включая имена студентов и курсы, на которые они зачислены.

Обновление записей (UPDATE)

Изменение существующих записей выполняется с помощью оператора `UPDATE`. Ниже описано, как обновить данные в таблицах `Students` и `Courses`.

```
-- Обновить сведения об электронной почте студента
UPDATE Students
SET Email = 'new.email@gitforgits.com'
WHERE Student_ID = 1;
-- Обновить описание курса
UPDATE Courses
SET Course_Description = 'An updated description of the course.'
WHERE Course_ID = 101;
```

С помощью этих операторов можно изменить адрес электронной почты студента и обновить описание курса.

Удаление записей (DELETE)

Удаление информации осуществляется с помощью оператора DELETE. Ниже приведены методы удаления записей из таблицы Enrollments и связанные с ними действия.

```
-- Удалить зачисление
DELETE FROM Enrollments
WHERE Enrollment_ID = 1;
-- Удаление информации о студенте и всех его зачислениях
DELETE FROM Students
WHERE Student_ID = 1;
-- Предполагается, что используется каскадный метод
-- CASCADE DELETE, в противном случае запись сначала нужно
-- вручную удалить из таблицы Enrollments
```

При удалении записи о студенте сначала убедитесь, что ограничение внешнего ключа для зачислений настроено на каскадное удаление. Если это не так, вручную удалите связанные со студентом зачисления, чтобы сохранить целостность ссылок.

Рекомендации по выполнению операций CRUD

Для операций, состоящих из нескольких этапов (например, удаление информации о студенте и его зачислениях), используйте транзакции, которые гарантируют успешное или неудачное завершение всех операций. Это обеспечивает целостность данных.

```
BEGIN TRANSACTION;
DELETE FROM Enrollments WHERE Student_ID = 1;
DELETE FROM Students WHERE Student_ID =1;
COMMIT;
```

Для поддержания базы данных в актуальном состоянии, а также чтобы приложение реагировало на запросы пользователей, необходимо выполнять все вышеперечисленные операторы.

Расширенные методы обработки запросов

В продолжение рассмотрения основных операций CRUD, с которыми мы познакомились ранее, обратимся к усовершенствованной технике запросов, которая позволяет более тонко и эффек-

тивно извлекать данные и манипулировать ими. Подзапросы, объединения и агрегатные функции языка SQL являются частью этих методов, позволяющих проводить более сложный анализ и извлечение данных. Основная цель этих методов — расширить функциональность базы данных системы управления университетом, добавив расширенные возможности запросов, выходящие за рамки базовых операций CRUD.

Подзапросы

Подзапросы — это запросы, вложенные «внутрь» основного SQL-запроса. Подзапросы часто используются в операторах `SELECT`, `FROM` или `WHERE`. Подзапросы могут использоваться для выполнения операций, требующих многоступенчатой фильтрации данных, или для обеспечения вычислений.

Пример программы: «Поиск курсов, на которые не зарегистрировался ни один студент»

Предположим, вам нужно найти курсы, на которые не записался ни один студент. Этого можно добиться с помощью подзапроса, исключающего курсы, записи о которых есть в таблице `Enrollments` (Зачисления).

```
SELECT Course_ID, Course_Name
FROM Courses
WHERE Course_ID NOT IN (SELECT Course_ID FROM Enrollments);
```

С помощью этого запроса мы сможем найти курсы, запись о которых ни разу не встречается в таблице `Enrollments`. Таким образом мы можем получить список курсов, на которые не зачислен ни один студент.

Объединения

Объединения используются для соединения нескольких строк из двух или более таблиц. Объединение происходит на основе связанного между этими таблицами столбца. Этот метод используется в запросах, где нужно объединить данные из нескольких связанных сущностей.

Пример программы: «Список студентов с информацией о курсах»

Чтобы получить подробный список студентов с указанием курсов, на которые они зачислены, используется оператор JOIN:

```
SELECT s.Name, s.Email, c.Course_Name
FROM Students s
JOIN Enrollments e ON s.Student_ID = e.Student_ID
JOIN Courses c ON e.Course_ID = c.Course_ID;
```

Этот запрос дает полное представление о том, кто из студентов на какие курсы зачислен. В запросе используется метод объединения для соединения данных из трех таблиц.

Агрегатные функции SQL

Агрегатные функции SQL выполняют вычисления, обрабатывая набор значений, и возвращают одно значение. Эти функции используются совместно с оператором GROUP BY, назначение которого — группировка строк, имеющих одинаковые значения в указанных столбцах, в сводные строки.

Пример программы: «Подсчет количества студентов, записанных на каждый курс»

Чтобы определить, насколько популярен тот или иной курс, можно подсчитать количество студентов, записанных на каждый курс.

```
SELECT c.Course_Name, COUNT(*) as Student_Count
FROM Courses c
JOIN Enrollments e ON c.Course_ID = e.Course_ID
GROUP BY c.Course_Name;
```

Этот запрос группирует результаты по названию курса и подсчитывает количество записей в таблице Enrollments для каждого курса, получая количество студентов по курсам.

Расширенная фильтрация с помощью оператора *HAVING*

Оператор HAVING используется в агрегатных функциях для фильтрации результатов, полученных с помощью оператора GROUP BY. В отличие от оператора WHERE, оператор HAVING умеет фильтровать агрегированные данные.

Пример программы: «Курсы с более чем 5 студентами»

Если вы хотите найти только те курсы, на которые записано более 5 студентов, сделайте следующее:

```
SELECT c.Course_Name, COUNT(*) as Student_Count
FROM Courses c
JOIN Enrollments e ON c.Course_ID = e.Course_ID
GROUP BY c.Course_Name
HAVING COUNT(*) > 5;
```

Этот запрос выводит список курсов, которые пользуются особой популярностью и на которые записано более 5 студентов.

Оконные функции

Оконные функции позволяют выполнять вычисления для набора строк таблицы, которые каким-то образом связаны с текущей строкой. Это похоже на агрегатные функции, но в оконных функциях строки не группируются в одну выходную строку.

Пример программы: «Распределение студентов по дате зачисления на каждый курс»

Предположим, вам нужно упорядочить информацию о студентах по дате их зачисления на каждый курс:

```
SELECT s.Name, c.Course_Name, e.Enrollment_Date,
RANK() OVER (PARTITION BY e.Course_ID ORDER BY e.Enrollment_Date)
as Enrollment_Rank
FROM Enrollments e
JOIN Students s ON e.Student_ID = s.Student_ID
JOIN Courses c ON e.Course_ID = c.Course_ID;
```

В этом запросе используется оконная функция, с помощью которой происходит распределение студентов внутри каждого курса на основе даты их зачисления, и при этом результаты в одну строку для каждого курса не собираются.

Такие методы позволяют выполнять более сложные запросы, что крайне важно для всестороннего анализа данных и составления отчетов в таких объемных приложениях, как система управления университетом. Благодаря этим методам становится возможным использовать более рациональный подход к выполнению запросов к базе данных, что приводит к получению наиболее полных сведений и повышению эффективности работы с данными.

Миграция баз данных и контроль версий

Вносимые со временем изменения в схему базы данных необходимо обрабатывать без потери информации. Именно здесь на помощь приходит миграция базы данных. Приложения, созданные в гибких средах разработки, часто требуют итеративных изменений в структуре базы данных, поэтому этот процесс крайне важен. Чтобы миграция базы данных выполнялась эффективно, необходимо отслеживать все изменения схемы таким образом, чтобы их можно было применять, откатывать, а также контролировать их версии вместе с кодом приложения.

> ***Примечание переводчика***
>
> *Миграция данных* — это перенос данных из одной вычислительной среды или системы хранения в другую.
>
> *Гибкая среда разработки* — это термин, используемый для описания итеративной разработки программного обеспечения. Итеративная разработка программного обеспечения сокращает жизненный цикл DevOps с помощью завершения работы в коротких сеансах, обычно называемых спринтами.
>
> *Итеративный подход* в разработке программного обеспечения — это выполнение работ параллельно с непрерывным анализом полученных результатов и корректировкой последующих этапов работы. Проект при таком подходе в каждой фазе развития проходит повторяющийся цикл PDCA: планирование — реализация — проверка — корректировка.

Основные сведения о миграции баз данных

Миграция баз данных обычно включает такие изменения, как добавление или удаление таблиц, изменение таблиц с добавлением или удалением столбцов, а также изменение индексов и ограничений. Эти изменения сохраняются в сценариях миграции — файлах, которые выполняются для изменения схемы базы данных. Эти сценарии не только обновляют схему, но и без потерь обеспечивают соответствие существующих данных новой схеме.

Для облегчения выполнения миграции баз данных существуют различные инструменты, каждый из которых совместим с разными типами баз данных и фреймворков. Для приложений на базе языка программирования Python, особенно для тех приложений, которые используют программную библиотеку SQLAlchemy, часто используются такие инструменты, как Alembic. Alembic представляет собой миграционный фреймворк для SQLAlchemy, который позволяет надежно и эффективно переносить базу данных.

Настройка Alembic

Чтобы настроить Alembic для запуска миграции баз данных, выполните следующие действия.

Установка Alembic

```
pip install alembic
```

Инициализация Alembic

Перейдите в каталог, в котором хранится ваш проект, и выполните команду:

```
alembic init alembic
```

Эта команда создает новый каталог под названием **alembic**, содержащий ваши сценарии миграции и файл конфигурации **alembic.ini**.

Настройка Alembic

Измените файл **alembic.ini**, указав строку подключения к вашей базе данных. Вам также нужно настроить файл **env.py** в каталоге **alembic**, чтобы определить, как инструмент с помощью программной библиотеки SQLAlchemy подключается к вашей базе данных.

```
from myapp.models import Base # предполагается, что ваши модели
                              # находятся в файле myapp/models.py
target_metadata = Base.metadata
```

Создание и применение миграций

После того как Alembic настроен, вы можете создавать и применять миграции.

Создание миграции

Чтобы автоматически сгенерировать новый сценарий миграции на основе различий между вашими моделями и текущей схемой базы данных, выполните команду:

```
alembic revision --autogenerate -m "Description of the change"
```

Эта команда добавит новый сценарий в папку **versions**, расположенную в каталоге **alembic**. Просмотрите этот сценарий, чтобы убедиться, что он точно отражает все необходимые изменения.

Редактирование сценария миграции

Иногда автоматическая генерация может не совсем точно отражать внесенные сложные изменения или специфические операции с базой данных. В таких случаях вручную отредактируйте сценарий миграции, находящийся в каталоге **versions**, чтобы исправить или уточнить операции.

Применение миграций

Чтобы обновить базу данных до последней версии, используйте команду:

```
alembic upgrade head
```

Эта команда применит к базе данных все запланированные ранее миграции. Для отката можно использовать команду:

```
alembic downgrade -1
```

Эта команда отменяет последнюю примененную миграцию.

Управление изменениями схемы базы данных

Управление изменениями схемы базы данных с помощью контроля версий включает в себя несколько перечисленных далее этапов.

Создание версий

Каждый сценарий миграции маркируется по времени и хранится в репозитории вместе с контролем версий и кодом вашего приложения. Такая синхронизация позволяет сопоставить каждое состояние кода приложения с соответствующим состоянием схемы базы данных.

Совместная работа

При командной работе миграции гарантируют, что все участники работают с одной и той же схемой базы данных. После внесения изменений, включающих новые миграции, все члены команды могут применить эти миграции к своим локальным базам данных, предназначенным для разработки.

Развертывание

Во время процесса развертывания миграции осуществляются как часть процесса развертывания. Этим гарантируется, что схема производственной базы данных совпадает со схемой, которую использует развернутая версия приложения.

Alembic и аналогичные инструменты позволяют легко управлять внесениями изменений в разных средах и сохранять согласованность кода приложения и схемы базы данных, что очень важно для выполнения безошибочной и безопасной миграции.

Интеграция баз данных с ORM на Python

Обзор ORM для Python

Python ORM — это связующее звено между кодом, написанным на языке Python, и базой данных. В результате для большинства операций отпадает необходимость в ручном написании SQL-запросов, что снижает вероятность атак с использованием SQL-инъекций и синтаксических ошибок. Наиболее популярными ORM для Python считаются программная библиотека SQLAlchemy, инструмент фреймворка Django ORM и Peewee. В этой книге мы рассмотрим программную библиотеку SQLAlchemy, которая по сравнению с остальными доступными вариантами обладает наибольшей функциональностью.

> ***Примечание переводчика***
>
> ORM (Object-Relation Mapping или объектно-реляционное отображение) — это общее название для фреймворков или библиотек, позволяющих автоматически связывать базу данных с кодом. ORM стараются скрыть существование базы данных настолько, насколько это возможно.
>
> Django ORM — это инструмент фреймворка Django, который позволяет взаимодействовать с базами данных, используя не SQL-запросы, а высокоуровневые методы Python. Django ORM относится к типу ORM, который реализует шаблон Active Record. Общая суть шаблона в том, что каждой таблице в приложении соответствует одна модель.
>
> Peewee — это легкий и быстрый ORM, предназначенный для разработчиков Python.

Обзор SQLAlchemy

Программная библиотека SQLAlchemy — наиболее функциональный и гибкий ORM из всех доступных в экосистеме Python. Эта библиотека обеспечивает полную поддержку реляционных баз данных и предлагает выбор между двумя различными парадигмами использования: SQLAlchemy Core и SQLAlchemy ORM.

SQLAlchemy Core

В основу SQLAlchemy Core заложено использование SQL-выражений в качестве объектов Python. SQLAlchemy Core работает

с таблицами, индексами, колонками и т. д. Библиотека позволяет с высокой степенью точности контролировать использование запросов SQL и хорошо подходит для выполнения сложных операций с базами данных, которые не укладываются в шаблон Active Record.

SQLAlchemy ORM

SQLAlchemy ORM — это интерфейс более высокого уровня, который позволяет работать с моделями и классами, связанными с таблицами базы данных. Этот интерфейс содержит систему отношений, которая может использоваться для автоматического сохранения изменений в объектах и связанных с ними других объектах.

Интеграция SQLAlchemy ORM с Python

Чтобы интегрировать SQLAlchemy ORM в приложение, написанное на языке Python, необходимо перед началом работы с записями в базе данных выполнить несколько шагов по настройке и использованию этой ORM.

Установка

Начните с установки SQLAlchemy с помощью менеджера пакетов pip.

```
pip install SQLAlchemy
```

Определение моделей

Моделями в SQLAlchemy называются классы, которые определяют структуру таблицы базы данных. Каждый атрибут класса соответствует столбцу таблицы базы данных.

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
Base = declarative_base()
class User(Base):
__tablename__ = 'users'
```

```
id = Column(Integer, primary_key=True)
name = Column(String)
email = Column(String)
# Создайте механизм, который будет хранить данные
# в локальной директории
engine = create_engine('sqlite:///mydatabase.db')
Base.metadata.create_all(engine)
```

Создание сессии

SQLAlchemy для управления операциями над объектами, связанными с базой данных, использует сессии. Сессия устанавливает все соединения с базой данных и представляет собой «зону ожидания» для всех объектов, которые вы загрузили или связали с ней за время ее существования.

```
Session = sessionmaker(bind=engine)
session = Session()
```

Выполнение операций с базой данных

Создание

```
new_user = User(name='John Doe', email='john@gitforgits.com')
session.add(new_user)
session.commit()
```

Чтение

```
user = session.query(User).filter_by(name='John Doe').first()
print(user.email)
```

Обновление

```
user.email = 'new.email@gitforgits.com' session.commit()
```

Удаление

```
session.delete(user)
session.commit()
```

Вы можете значительно упростить взаимодействие баз данных и своих приложений, написанных на языке Python, интегрировав в них SQLAlchemy. Благодаря тому, что библиотека не использует необработанные SQL-запросы, код становится более простым

и удобным в обслуживании. Кроме того, библиотека предлагает такие мощные инструменты, как автоматическая миграция схемы, управление отношениями и реализация модулей работы, которые могут справиться даже с самыми сложными моделями данных.

Стратегии кеширования для оптимизации баз данных

Для повышения скорости работы приложений, взаимодействующих с базами данных, наиболее эффективным методом является кеширование. Этот метод предполагает сокращение необходимости многократного обращения к базе данных за одними и теми же данными. Осуществляется это с помощью временного хранения копий полученных данных в быстродействующем оборудовании, например в оперативной памяти. Кеширование значительно снижает нагрузку на базу данных, одновременно сокращая время отклика в пиковые моменты использования. Здесь мы рассмотрим различные стратегии кеширования, которые используются веб-приложениями для улучшения взаимодействия с базами данных.

Типы кеширования

Кеширование результатов

Речь идет о хранении результатов запросов, ранее полученных при обращении к базе данных. Когда запрос выполняется в первый раз, набор его результатов сохраняется в кеше. Результирующий набор данных для последующих таких же запросов будет получен непосредственно из кеша, полностью минуя фазу запроса к базе данных.

Кеширование объектов

При объектном кешировании вместо необработанных результатов запросов в кеше сохраняются объекты данных. Этот метод очень хорошо работает в приложениях, использующих ORM, когда взаимодействие с данными осуществляется через представ-

ления объектов. Кеширование объектов может помочь быстро получить объекты сущностей без необходимости восстановления их из результатов запросов к базе данных.

Примечание переводчика

Объекты данных — это местоположение или область хранения, содержащая набор атрибутов или групп значений. Они действуют как аспект, характеристика, качество или дескриптор объекта.

Объекты сущностей — это классы, которые инкапсулируют бизнес-модель, включая правила, данные, взаимосвязи и поведение сохранения, для элементов, используемых в бизнес-приложении.

Инкапсуляция — это способ организации кода, который позволяет держать код и данные вместе и в безопасности от случайных изменений.

В программировании сущность — это как коробочка, в которой хранятся данные и правила для них. Например, персонаж в игре, у которого есть имя, характеристики и способности. Это не просто данные, это данные с инструкциями, описывающими, на что эти данные способны. Сущность решает проблему организации и управления данными.

В программировании, сущности и объекты часто путают, но между ними есть ключевые различия. Объекты — это экземпляры классов, которые могут содержать данные и методы для работы с этими данными. Сущности же — это особый тип объектов, которые имеют уникальный идентификатор (ID) и представляют собой конкретные данные в предметной области.

Проще говоря, если объект — это любая книга на полке, то сущность — это конкретная книга с уникальным номером, которую вы можете взять в руки. Это различие важно, потому что сущности помогают управлять уникальными данными и обеспечивают их целостность в программе или базе данных.

Кеширование планов запросов

SQL-базы данных часто кешируют планы выполнения запросов, которые представляют собой стратегии, используемые движком базы данных для выполнения запросов.

Кеширование планов выполнения запросов позволяет избежать накладных расходов на оптимизацию запросов при последующих выполнениях одной и той же операции, что ускоряет процесс его выполнения.

Реализация кеширования в веб-приложениях

Redis в качестве кеша

Redis — популярный высокопроизводительный инструмент с богатым набором структур данных, позволяющий реализовать кеширование в веб-приложениях. Redis полностью находится в оперативной памяти и может использоваться как кеш для временного хранения результатов и объектов.

Ниже показана базовая настройка Redis, чтобы его можно было использовать в приложении, созданном на языке Python:

1. Установите Redis и запустите службу Redis на своей машине или воспользуйтесь облачным сервисом Redis.
2. Для внедрения кеширования в ваше приложение воспользуйтесь библиотекой `redis-py`.

```
pip install redis
import redis
# Подключение к локальному экземпляру Redis
r = redis.Redis(host='localhost', port=6379,db=0)
# Пример установки и получения значения кеша
r.set('foo', 'bar')
value = r.get('foo')
print(value) # Выходные данные: b'bar'
```

Стратегия кеширования

1. Сохраните результат выполнения запроса в Redis. В качестве ключа для хранилища Redis используйте саму строку запроса.

```
def get_user(user_id):
# Проверяем, появился ли результат в кеше
result = r.get(f'user_{user_id}')
if result is None:
# Если результата в кеше нет, извлекаем его из базы данных
# и копируем
user = query_database(f"SELECT * FROM users WHERE id = {user_id}")
r.set(f'user_{user_id}', serialize(user))
return user
return deserialize(result)
```

2. Для предотвращения обслуживания устаревших данных очень важно своевременно аннулировать кеш. При изменении данных аннулируйте или обновляйте запись в кеше.

```
def update_user(user_id, new_data):
# Обновление базы данных
update_database(f"UPDATE users SET data = {new_data}
WHERE id = {user_id}")
# Аннулирование кеша
r.delete(f'user_{user_id}')
```

Рекомендации по эффективному кешированию

- Выберите степень детализации кеширования. При уменьшении размеров (например, кеширование отдельных объектов) использование кеша станет более эффективным, но при этом усложнится процедура аннулирования хранящихся в кеше данных.
- Установите оптимальное время жизни кеша, чтобы соблюсти баланс между актуальностью данных и производительностью. Используйте настройки времени ожидания (TTL) для записей кеша, чтобы данные не слишком устаревали.
- Согласуйте кеш и базу данных. Если не будет правильного взаимодействия, это может стать одной из самых больших проблем в управлении кешем, требующей надежных стратегий для аннулирования и обновления кеша.
- Производительность вашего приложения может быть улучшена, а нагрузка на базу данных, наоборот, уменьшена, если вы примените стратегии кеширования. Решения для управления кешем, такие как Redis, позволяют приложениям легче масштабироваться и быстрее реагировать на запросы благодаря тому, что часто запрашиваемые данные хранятся в легкодоступном месте памяти.

Резюме

В этой главе мы уделили особое внимание комплексным методам управления базами данных и стратегиям оптимизации, которые имеют решающее значение для эффективной разработки внут-

ренних частей приложения. Сначала мы познакомились с реляционными базами данных MySQL и PostgreSQL. Здесь нами были рассмотрены основные возможности каждой базы данных и сценарии, с помощью которых вам будет легче выбрать подходящую базу данных для вашего проекта. Далее мы перешли к базам данных NoSQL, особо выделив MongoDB по причине ее адаптивности и скорости в управлении теми видами операций с массивными данными, которые встречаются в современных приложениях.

Затем мы поговорили об основах проектирования баз данных, при этом подчеркивается важность создания хорошо структурированных баз данных, что обеспечит целостность данных и производительность запросов. Чтобы помочь в разработке и реализации надежных схем баз данных, особое внимание уделяется таким понятиям, как нормализация и использование моделей сущностей и отношений.

В дополнение мы рассмотрели более сложные операции CRUD, показав, как с их помощью можно улучшить взаимодействие с базой данных. Чтобы усовершенствовать навыки управления динамическими данными в системе управления университетом, мы рассмотрели, как можно создать, прочитать, обновить и удалить данные.

Затем последовал обзор передовых методов составления запросов, включающий новые способы получения более сложных данных. Мы познакомились с несколькими методами, облегчающими сложные манипуляции с данными и их анализ. В эти методы входят подзапросы, объединения, агрегатные функции и оконные функции.

В завершение главы нами была подробно рассмотрена миграция баз данных и контроль версий. Здесь мы обратили ваше внимание на важность миграции данных и контроля версий при управлении изменениями схемы базы данных, чтобы с течением времени не потерять необходимую информацию. Особое внимание было уделено эффективности таких инструментов, как Alembic, с помощью которого можно выполнить мониторинг, модифицировать или откатить внедренные ранее модификации баз данных еще во время разработки приложения.

И в конце главы мы рассказали о стратегиях кеширования, которые могут помочь оптимизировать работу баз данных. Здесь нами были рассмотрены различные стратегии кеширования, такие как кеширование объектов и кеширование результатов, и показано, как эти стратегии можно использовать с помощью таких инструментов, как Redis. Так мы снизим нагрузку на базу данных и значительно повысим производительность приложений.

В целом глава была посвящена созданию прочного фундамента знаний по эффективному управлению базами данных, что очень важно для обеспечения эффективной и бесперебойной работы приложений в соответствии с их назначением, а также их быстрого и простого масштабирования.

ГЛАВА 6
Асинхронное программирование в Python

Введение

В этой главе мы покажем на практике применение асинхронного программирования в среде Python и рассмотрим принципы работы таких приложений. Чтобы помочь читателям приобрести навыки написания эффективных и масштабируемых приложений, в этой главе мы подробно расскажем о возможностях асинхронного программирования в Python.

Первая остановка на этом пути — *раздел «Общее описание асинхронного программирования»*, в котором мы рассмотрим основные принципы асинхронного программирования, в чем состоит преимущество такого подхода и ситуации, в которых этот способ превосходит более традиционные синхронные методы программирования. Так мы заложим основу для более широкого понимания возможностей и применения этого метода программирования.

В следующем *разделе «Основы Asyncio»*, вы узнаете, как использовать синтаксис async/await для написания асинхронного кода на Python. В этом разделе рассматриваются циклы событий, задачи и корутины (сопрограммы), необходимые для выполнения асинхронных операций, а также другие основные компоненты `asyncio`.

> ***Примечание переводчика***
>
> *Корутины* (*сопрограммы*) — это функции, работающие асинхронно, т. е. по очереди. В нужный момент исполнение первой функции приостанавливается с сохранением всех ее свойств, чтобы запустился другой код. Когда управление возвращается к первой функции, она продолжает работу.

Корутины нужны для:

- создания асинхронных приложений, которые могут выполнять несколько действий одновременно;
- гибкой и удобной реализации многозадачности;
- лучшего контроля при переключении между разными задачами;
- снижения нагрузки на аппаратные ресурсы устройства.

Темы, затронутые в *разделе «Разработка асинхронных веб-приложений»*, посвящены использованию `asyncio` в различных условиях веб-разработки. В приведенном здесь примере мы покажем вам, как повысить скорость и отзывчивость ваших веб-приложений за счет управления структурированным сетевым кодом и запросами ввода-вывода.

Следующий *раздел «Асинхронный доступ к базам данных»* посвящен операциям с базой данных, выполняемым в разное время. Чтобы веб-приложения были отзывчивыми, необходимо реализовать асинхронный доступ к базам данных. Это позволит выполнять неблокирующие запросы и обновления базы данных, о которых пойдет речь в этом разделе.

Для тех кто не знаком с WebSockets и возможностями их использования в условиях асинхронного взаимодействия, был написан *раздел «Внедрение веб-сокетов»*. Постоянное соединение с малым временем ожидания между клиентом и сервером идеально подходит для веб-приложений, работающих в режиме реального времени, поскольку такое соединение значительно облегчает работу пользователей.

Когда речь заходит о создании эффективного, хорошо структурированного и легко поддерживаемого асинхронного кода, *раздел «Передовой опыт и шаблоны Async»* становится очень полезным ресурсом. Этот раздел призван помочь разработчикам максимально использовать асинхронные возможности языка Python, избегая при этом типичных ошибок.

И наконец, в *разделе «Отладка асинхронных приложений»* мы расскажем о методах решения проблем, связанных с асинхронными приложениями, которые, как известно, трудно отлаживать из-за тонкостей выполнения параллельных операций.

Общее описание асинхронного программирования

При асинхронном программировании одна операция не блокирует выполнение другой операции. Это один из способов управления для выполнения параллельных операций в программировании. Если ожидание завершения действия одной операции может привести к потере процессорного времени, которое могло бы быть использовано для выполнения других задач, такой подход оказывается очень полезным, особенно в процедурах, связанных с вводом-выводом и с большим временем ожидания.

Что такое асинхронное программирование?

При асинхронном программировании все операции выполняются в режиме отсутствия блокировки. В отличие от синхронного программирования, где код выполняется последовательно и каждая строка должна дождаться завершения выполнения предыдущей строки, асинхронный код позволяет программе обрабатывать несколько операций, инициируя одну операцию и переходя к другой до завершения первой.

Асинхронное программирование для разработки внутренних компонентов

При работе приложений в их внутренних системах довольно часто выполняются операции, которые могут замедлить выполнение вашего кода: например, запросы к базе данных, операции ввода-вывода файлов или запросы к другим веб-сервисам. При традиционном синхронном программировании, прежде чем перейти к следующей операции, сервер ожидает завершения предыдущих операций. Это может привести к неэффективному использованию ресурсов и низкой производительности, особенно в условиях высокой нагрузки, когда одновременно обрабатывается множество запросов.

Решение этой проблемы — асинхронное программирование, позволяющее серверу выполнять другие задачи в ожидании завер-

шения операций ввода-вывода. Это повышает эффективность и скорость отклика приложения, позволяя ему обслуживать больше пользователей без добавления дополнительного оборудования.

Примеры использования асинхронного программирования

Веб-серверы

Для обработки веб-запросов наиболее подходят асинхронные серверы. Эти серверы могут управлять тысячами соединений, что делает их идеальными для работы веб-приложений в режиме реального времени. Например, среда для выполнения кода Node.js, изначально созданного на основе JavaScript-движка V8, была разработана с использованием неблокирующей, управляемой событиями архитектурой, что позволяет эффективно обрабатывать множество одновременных соединений.

Обработка данных

Для повышения скорости обработки данных рекомендуется использовать возможности асинхронного программирования. Например, при обработке с помощью методов асинхронного программирования больших объемов данных из базы данных или API можно получать и обрабатывать данные параллельно, а не ждать последовательного завершения каждого запроса или вызова API.

Работа приложения в режиме реального времени

Такие программы, как чат-приложения, уведомления в режиме реального времени или панели данных, отображающих информацию в реальном времени, получают от применения метода асинхронного программирования значительные преимущества. Например, приложение для чата может обрабатывать отправку и получение сообщений, не прерывая работу пользовательского интерфейса, обеспечивая этим бесперебойную работу с минимальными интервалами для отзыва.

Архитектура микросервисов

В архитектуре микросервисов различным сервисам может потребоваться взаимодействовать друг с другом по протоколу HTTP. Асинхронные HTTP-запросы позволяют этим сервисам работать самостоятельно, не ожидая ответов от других запросов, тем самым повышая общую производительность системы.

Как работает асинхронное программирование?

В Python асинхронное программирование обычно реализуется с помощью библиотеки `asyncio`, которая предоставляет основу для запуска асинхронных задач и обработки обратных вызовов. Далее приведен краткий обзор ее структуры.

Цикл событий

Сердцем `asyncio` является цикл обработки событий, который отвечает за управление и распределение выполнения различных задач. Цикл событий отслеживает все запущенные задачи и выполняет их, когда они будут готовы, управляя их состояниями в неблокирующем режиме.

Корутины (сопрограммы или асинхронные функции)

Корутина или сопрограмма — это асинхронная функция, которая может приостановить свое выполнение до получения результата другой асинхронной операции, а также на некоторое время косвенно передать управление другим асинхронным функциям. Корутины объявляются с помощью ключевых слов `async def`, а их выполнение приостанавливается с помощью ключевого слова `await`.

Задачи и фьючерсы

Фьючерсы — это объекты, которые связывают обратный вызов с результатом выполнения функции. В `asyncio` задачи — это подтип фьючерсов, которые используются для одновременного планирования сопрограмм. Когда сопрограмма с помощью таких функций, как `asyncio.create_task()`, преобразуется в задачу, цикл

обработки событий начинает заботиться об управлении выполнением этой задачи.

Пример программы: «Асинхронное выполнение»

Ниже приведен простой пример, иллюстрирующий процесс асинхронного выполнения задач в Python:

```
import asyncio
async def main():
print('Hello')
await asyncio.sleep(1)
print('World')
# Запуск асинхронной функции
asyncio.run(main())
```

В приведенном коде метод `asyncio.run(main())` запускает основную сопрограмму и планирует ее выполнение. Оператор `await asyncio.sleep(1)` приостанавливает выполнение сопрограммы, позволяя циклу обработки событий выполнить другие задачи (если таковые были) в течение 1-секундного периода ожидания, а затем возобновляет выполнение сопрограммы.

Повышение отзывчивости и производительности приложений, а также оптимизация потребления ресурсов (особенно для задач, связанных с вводом-выводом) — все это результаты, которых можно достичь, используя данную функцию. Умение использовать библиотеку asyncio из Python для реализации асинхронных операций теперь является обязательным условием для современных инженеров по разработке внутренних компонентов приложений.

Основы asyncio

Библиотека **asyncio** в Python предназначена для создания механизма написания кода с одновременной обработкой данных с помощью синтаксиса `async` и `await`. Эта библиотека предназначена для асинхронного программирования на языке Python и позволяет выполнять различные асинхронные операции без блокировки основного выполняемого потока данных. Знание основ **asyncio** и

ключевого синтаксиса `async` и `await` помогает значительно повысить эффективность наших приложений, написанных на Python, особенно когда речь идет о ситуациях, связанных с вводом-выводом и интенсивным использованием данных.

Синтаксис *async*

Ключевое слово `async` используется для объявления функции как сопрограммы (coroutine), которая приостанавливает свое выполнение до получения результата другой асинхронной операции и может на некоторое время косвенно передавать управление другим корутинам.

```
async def fetch_data():
# Функция, которая получает данные в асинхронном режиме
data = await some_async_operation()
return data
```

Синтаксис *await*

Ключевое слово `await` используется в асинхронной функции (async function) для приостановки выполнения сопрограммы до завершения ожидаемой задачи. Ключевое слово `await` может использоваться только в асинхронных функциях.

Этот прием позволяет выполнять другие задачи во время, когда выполняемая сопрограмма остановлена, эффективно используя цикл событий.

```
async def process_data():
data = await fetch_data()
print("Данные обработаны:", data)
```

Структура программы asyncio

Ниже показана базовая структура программы asyncio:

```
import asyncio
async def main():
print('Start')
```

```
await asyncio.sleep(1) # Имитирует асинхронную операцию
                       # ввода-вывода
print('Finish')
asyncio.run(main())
```

В приведенном примере программы метод `asyncio.run(main())` используется для выполнения главной корутины (сопрограммы), которая управляет другими асинхронными функциями и задачами.

Пример программы: «Интеграция библиотеки asyncio в приложение для университета»

Давайте интегрируем модуль `asyncio` в систему управления университетом, уделив особое внимание смоделированному сценарию, в котором требуются асинхронные операции, такие как получение информации о студентах и асинхронная обработка данных о зачислении.

```
import asyncio
# Имитация вызова базы данных
async def get_student(student_id):
print(f "Получение данных о студенте {student_id}")
await asyncio.sleep(2) # Имитация времени ответа базы данных
return { "id": student_id, "name": "John Doe"}
# Имитация вызова базы данных
async def enroll_student_in_course(student, course_id):
print(f "Зачисление {студент[„name"]} на курс {course_id}")
await asyncio.sleep(1) # Имитация обработки зачисления
return True
# Главная сопрограмма, которая управляет другими сопрограммами
async def main():
student_id = 123
course_id = 101
# Получение информации о студенте
student = await get_student(student_id)
# Зачислить студента на курс
enrolled = await enroll_student_in_course(student, course_id)
```

```
if enrolled:
print(f"{студент[„name"]} был зачислен на курс {course_id}")
# Запуск главной сопрограммы
asyncio.run(main())
```

В приведенном выше примере программы:

- Получение данных о студентах — метод `get_student` имитирует получение данных о студентах из базы данных. Это асинхронная функция, которая, имитируя работу базы данных, ждет две секунды.
- Зачисление на курс — метод `enroll_student_in_course` имитирует зачисление студента на курс. В течение одной секунды происходит ожидание, имитирующее время выполнения процесса.
- Согласование — главная сопрограмма организует выполнение указанных выше задач. Она ждет, пока будут получены данные о студенте, и затем приступает к зачислению студента на курс.

Разработчики, специализирующиеся на написании программ на языке Python, особенно программ, работающих в режиме ввода-вывода или требующих большого количества одновременной обработки данных, могут извлечь значительную пользу, изучив этот раздел, описывающий основы работы с модулем `asyncio`, и внедряя его в свои приложения. Умение разрабатывать асинхронные программы на Python просто необходимо в современном мире программирования, поскольку позволяет создавать эффективный и понятный код с использованием модулей `async` и `await`.

Разработка асинхронных веб-приложений

Когда речь заходит об асинхронном программировании, основным преимуществом веб-фреймворка FastAPI считается встроенная в него возможность обработки маршрутов асинхронной передачи данных. Функция явного использования модулей `async` и `await` в реализациях конечных точек значительно повышает эффективность операций, связанных с вводом-выводом.

Пример программы: «Асинхронное университетское приложение»

Сейчас мы усовершенствуем систему управления асинхронным веб-сервером университета, которая сможет обрабатывать данные о студентах и записывать их на курсы, не ограничивая время отклика сервера.

Настройка и установка

Во-первых, убедитесь, что у вас установлены веб-фреймворк FastAPI и ASGI-сервер **uvicorn**:

```
pip install fastapi uvicorn
```

Определение конечных точек асинхронной связи

Мы создадим простое приложение на основе веб-фреймворка FastAPI, в котором будут использоваться асинхронные конечные точки для получения данных о студентах и зачисления их на курсы.

```
from fastapi import FastAPI, HTTPException
import asyncio
app = FastAPI()
# Имитация операций с базой данных
async def get_student_db(student_id):
await asyncio.sleep(1) # Имитируем операцию с базой данных
return { "id": student_id, "name": "John Doe"}
async def enroll_student_db(student_id, course_id):
await asyncio.sleep(1) # Имитация операции с базой данных
return { "student_id": student_id, "course_id": course_id}
@app.get("/students/{student_id}")
async def get_student(student_id: int):
student = await get_student_db(student_id)
if not student:
raise HTTPException(status_code=404, detail="Студент не найден")
return student
@app.post("/enroll/{student_id}/{course_id}")
async def enroll_student(student_id: int, course_id: int):
enrollment = await enroll_student_db(student_id, course_id)
```

```
if not enrollment:
raise HTTPException(status_code=404, detail="Зачисление не удалось")
return { "message": "Студент успешно зачислен", „enrollment": enrollment}
if __name__ == "__main__":
import uvicorn
uvicorn.run(app, host="127.0.0.1", port=8000)
```

В приведенном выше примере программы:

- `get_student_db` и `enroll_student_db` — это асинхронные функции, имитирующие операции с базой данных. Для имитации задержки операций с базой данных эти функции используют оператор `asyncio.sleep`;
- конечная точка `get_student` получает данные о студентах в асинхронном режиме, не блокируя другие запросы. Также конечная точка `enroll_student` обрабатывает зачисление студента на курс в асинхронном режиме;
- приложение запускается с помощью ASGI-сервера **uvicorn**, назначение которого — выполнение кода в асинхронном режиме. Это позволяет веб-фреймворку FastAPI обрабатывать несколько запросов одновременно, что значительно повышает производительность при интенсивной нагрузке;
- использование библиотеки **asyncio** с веб-фреймворком FastAPI особенно эффективно в приложениях, где такие операции, как доступ к информации о студентах и обработка результатов зачисления, могут получить преимущество от неблокирующего ввода-вывода.

Асинхронный доступ к базам данных

Описание асинхронного доступа к базам данных

Традиционные синхронные операции с базами данных блокируют поток выполнения до тех пор, пока сервер базы данных не ответит. Это может привести к снижению производительности, особенно при высоких нагрузках, когда несколько пользователей выполняют операции с базой данных одновременно. Асинхрон-

ный доступ к базе данных решает эту проблему, позволяя выполнять другие задачи в ожидании завершения операций с базой данных.

Хотя многие традиционные драйверы баз данных Python сразу после установки не поддерживают работу с модулем `asyncio`, существует несколько библиотек, предназначенных для работы с модулем `asyncio` и обеспечивающих асинхронное взаимодействие с базами данных. Для баз данных SQL интерфейсы, совместимые с модулем `asyncio`, предоставляют библиотеки `aiomysql` (для MySQL) и `aiopg` (для PostgreSQL). В данном примере, предполагая, что мы используем базу данных PostgreSQL, мы будем использовать библиотеку `aiopg`.

Настройка и установка

Сначала нужно установить необходимый пакет `aiopg` — библиотеку, совместимую с `asyncio` для системы управления базами данных PostgreSQL:

```
pip install aiopg
```

Интеграция асинхронных операций с базами данных в веб-фреймворке FastAPI

Чтобы продемонстрировать выполнение асинхронных операций с базами данных, внесем в предыдущий пример университетского приложения изменения, чтобы использовать библиотеку `aiopg` для получения данных о студентах и обработки зачислений.

Конфигурация

Для начала с помощью библиотеки `aiopg` установите соединение с базой данных. В веб-фреймворке FastAPI распространена практика обработки пула соединений с базой данных и управления сеансами при запуске и завершении работы приложения:

```
from fastapi import FastAPI
import aiopg
app = FastAPI()
```

```
dsn = 'dbname=mydatabase user=myuser password=mypassword
host=127.0.0.1'
db_pool = None
@app.on_event("startup")
async def startup():
global db_pool
db_pool = await aiopg.create_pool(dsn)
@app.on_event("shutdown")
async def shutdown():
global db_pool
if db_pool:
db_pool.close()
await db_pool.wait_closed()
```

Функции асинхронной базы данных

Настроив пул соединений, вы можете заняться написанием асинхронных функций для взаимодействия с базой данных. Эти функции будут использовать соединения из пула и выполнять SQL-запросы асинхронно.

```
async def get_student_db(student_id):
async with db_pool.acquire() as connection:
async with connection.cursor() as cursor:
await cursor.execute("SELECT * FROM students WHERE id = %s",
(student_id,))
student_record = await cursor.fetchone()
if student_record:
return {
"id": student_record[0],
"name": student_record[1],
"email": student_record[2]
}
return None
async def enroll_student_db(student_id, course_id):
async with db_pool.acquire() as connection:
async with connection.cursor() as cursor:
await cursor.execute(
"INSERT INTO enrollments (student_id, course_id) VALUES (%s, %s)",
(student_id, course_id)
)
```

```
await connection.commit()
return True
```

Использование асинхронных функций в маршрутах

Измените обработчики маршрутов FastAPI, чтобы задействовать данные асинхронные функции для работы с базой данных:

```
@app.get("/students/{student_id}")
async def get_student(student_id: int):
student = await get_student_db(student_id)
if not student:
raise HTTPException(status_code=404, detail=" Студент не найден")
return student
@app.post("/enroll/{student_id}/{course_id}")
async def enroll_student(student_id: int, course_id: int):
success = await enroll_student_db(student_id, course_id)
if not success:
raise HTTPException(status_code=500, detail="Зачисление прошло
неудачно")
return {"message": " Студент успешно зачислен"}
```

Поскольку данный метод исключает блокировку веб-сервера на время ожидания завершения операций с базой данных, этот метод особенно полезен в тех случаях, когда приложение должно обрабатывать большое количество одновременных действий при работе с базой данных.

Внедрение веб-сокетов

WebSocket — что это?

Веб-сокеты (от *англ.* WebSocket) с помощью одного долговременного соединения обеспечивают двустороннюю связь между клиентом и сервером. В отличие от HTTP-запросов, которые нестатичны, а порты закрываются сразу после получения ответа, соединения по протоколу WebSocket остаются открытыми на протяжении всего сеанса связи, что позволяет передавать данные в реальном времени и взаимодействовать напрямую между собой. Протокол WebSocket особенно хорошо работает с приложениями,

требующими постоянного обновления данных или взаимодействия в режиме реального времени. Это приложения для мгновенного обмена сообщениями, ленты новостей в режиме онлайн или интерактивные игры.

Веб-сокеты позволяют отправлять и получать сообщения между браузером и сервером без повторного открытия соединения, что уменьшает время задержки и накладные расходы. Непрерывное состояние соединения инициируется «рукопожатием» HTTP, после чего осуществляется переход с протокола HTTP на протокол WebSocket.

> ***Примечание переводчика***
>
> WebSocket (веб-сокет) — это независимый протокол связи, работающий поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером. Если говорить проще — это двусторонний протокол, который позволяет клиенту установить связь с сервером.
>
> SSL/TLS Handshake или «рукопожатие» между сервером и клиентом. Проще говоря, это идентификация друг друга. «Рукопожатие» происходит во время установления HTTPS-соединения внутри зашифрованного туннеля SSL/TLS, который гарантирует безопасность как серверу, так и клиенту. После успешной идентификации генерируется секретный сеансовый ключ, который обеспечивает защищенную связь. Этот ключ используется одновременно как для шифрования, так и для дешифрования передаваемых данных.

Реализация WebSocket в FastAPI

Веб-фреймворк FastAPI предоставляет встроенную поддержку протокола WebSocket, что упрощает реализацию функций связи в реальном времени. Для сценария университетского приложения рассмотрим реализацию протокола WebSocket, позволяющего в реальном времени сообщать администраторам или студентам о зачислении на курсы.

Настройка и базовая конечная точка WebSocket

Для начала мы с помощью веб-фреймворка FastAPI создадим базовую конечную точку для WebSocket. Эта конечная точка будет

прослушивать входящие соединения и отправлять сообщения клиенту.

Установка FastAPI и Uvicorn

Если приложение еще не запущено, убедитесь, что у вас установлены веб-фреймворк FastAPI и ASGI-сервер Uvicorn, с помощью которых можно запускать приложение.

```
pip install fastapi uvicorn
```

Определение конечной точки WebSocket

С помощью веб-фреймворка FastAPI создайте новое приложение и определите маршрут, который будет обрабатывать все входящие соединения через WebSocket.

```
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
app = FastAPI()
@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
await websocket.accept()
try:
while True:
data = await websocket.receive_text()
await websocket.send_text(f "Сообщение получено: {data}")
except WebSocketDisconnect:
print("Клиент отключен")
```

Эта конечная точка устанавливает соединение WebSocket по маршруту `/ws`. Метод `websocket.accept()` принимает входящее соединение. Затем сервер переходит в цикл, в котором он прослушивает сообщения (`receive_text()`) и передает каждое полученное сообщение обратно клиенту (`send_text()`).

Обновление записей в режиме реального времени

Чтобы наглядно показать, как это работает на практике, давайте усовершенствуем протокол WebSocket, чтобы он уведомлял подключенных клиентов о каждом новом зачислении. Такая возможность может быть полезна для обновления в реальном времени на панели администратора.

Сохранение соединений WebSocket

Если у вас есть несколько активных соединений WebSocket, храните информацию о них, используя, например, список или более сложный класс менеджера, который обрабатывает соединения и рассылает сообщения всем подключенным клиентам.

```
class ConnectionManager:
def __init__(self):
self.active_connections: List[WebSocket] = []
async def connect(self, websocket: WebSocket):
await websocket.accept()
self.active_connections.append(websocket)
def disconnect(self, websocket: WebSocket):
self.active_connections.remove(websocket)
async def broadcast(self, message: str):
for connection in self.active_connections:
await connection.send_text(message)
manager = ConnectionManager()
@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
await manager.connect(websocket)
try:
while True:
data = await websocket.receive_text()
await manager.broadcast(f"Новое зачисление: {data}")
except WebSocketDisconnect:
manager.disconnect(websocket)
print("Связь с клиентом разорвана")
```

Интеграция с процессом регистрации

Измените конечную точку регистрации так, чтобы она передавала сообщение всем подключенным клиентам WebSocket каждый раз, когда новый студент записывается на курс.

```
@app.post("/enroll/{student_id}/{course_id}")
async def enroll_student(student_id: int, course_id: int):
# Здесь предполагается логика зачисления
success = await enroll_student_db(student_id, course_id)
```

```
if success:
await manager.broadcast(f "Студент {student_id} зачислен на курс
{course_id}")
return {"Сообщение": "Студент успешно зачислен"}
raise HTTPException(status_code=500, detail="Зачисление не удалось")
```

Благодаря применению протокола WebSockets нам удалось создать более привлекательную и быстро реагирующую административную панель, которая используется в университетском приложении для информирования клиентов о наборе на курсы в режиме реального времени.

Передовой опыт и шаблоны Async

Обработка ошибок в асинхронном коде

В процессе программирования асинхронных операций, особенно при использовании таких фреймворков, как FastAPI, и модулей, подобных `asyncio`, следует придерживаться определенных правил и шаблонов. Это может значительно повысить надежность, ремонтопригодность и производительность ваших приложений. Асинхронные операции часто усложняют обработку ошибок из-за их неблокирующего свойства и объединения операций в цепочки. Правильная обработка ошибок гарантирует, что ваше приложение останется надежным и удобным для пользователя.

Пример программы: «Корректная обработка ошибок базы данных»

Подумайте о внедрении метода обработки ошибок для операций с базой данных, связанных с проблемами подключения или несоответствием данных.

```
async def get_student(student_id: int):
try:
student = await database.fetch_student(student_id)
return student
except ConnectionError:
# Обработка ошибок, специфичных для соединения
logger.error("Не удалось подключиться к базе данных.")
```

```
raise HTTPException(status_code=500, detail="Не удалось подключиться
к базе данных")
except Exception as e:
# Обработка других видов ошибок
logger.error(f "Произошла ошибка: {e}")
raise HTTPException(status_code=500, detail="Произошла внутренняя
ошибка")
```

Управление параллельными процессами и отмена задач

Правильное выполнение параллельных операций крайне важно для работы асинхронных приложений. С помощью такой обработки можно предотвратить утечки памяти и обеспечить эффективное управление ресурсами. Одним из элементов такой работы является отмена задач, которые могут долго выполняться или зависнуть.

Пример программы: «Отмена устаревших запросов к базе данных»

При выполнении длительного запроса вам может понадобиться реализовать функцию тайм-аута или отмены выполнения операции.

```
from asyncio import TimeoutError
async def fetch_data_with_timeout():
try:
return await asyncio.wait_for(database.fetch_large_data_set(),
timeout=10.0)
except TimeoutError:
logger.warning("Запрос к базе данных был отменен на основании
истечения установленного времени ожидания.")
return None
```

> ***Примечание переводчика***
>
> Функция тайм-аута в Python — это инструмент, который обеспечивает выполнение функции в заданном временном интервале. Если время исполнения функции превышено, вызов функции прерывается.

Использование менеджеров контекста для управления ресурсами

Менеджеры контекста крайне полезны в асинхронном программировании для управления такими ресурсами, как соединения с базой данных или файлами. Они обеспечивают правильную очистку ресурсов после их использования, предотвращая утечку ресурсов.

Пример программы: «Асинхронный менеджер контекста для подключения к базе данных»

```
from contextlib import asynccontextmanager
@asynccontextmanager
async def get_database_connection():
conn = await database.connect()
try:
yield conn
finally:
await conn.close()
async def use_database():
async with get_database_connection() as conn:
result = await conn.execute("SELECT * FROM students")
data = await result.fetchall()
return data
```

Разделение и модульное построение кода

В асинхронном программировании разделение компонентов и модульное построение кода может помочь справиться со сложностями, особенно при масштабировании приложений.

Пример программы: «Модульная обработка WebSocket»

Вместо того чтобы обрабатывать соединения WebSocket непосредственно в функциях конечных точек, вы можете выделить управление WebSocket в отдельный модуль или класс.

```
class WebSocketManager:
def __init__(self):
self.active_connections = []
```

```
async def connect(self, websocket):
await websocket.accept()
self.active_connections.append(websocket)
async def broadcast(self, message):
for websocket in self.active_connections:
await websocket.send_text(message)
async def disconnect(self, websocket):
self.active_connections.remove(websocket)
# Использование в маршрутах FastAPI
websocket_manager = WebSocketManager()
@app.websocket("/ws/{user_id}")
async def websocket_endpoint(websocket: WebSocket, user_id: int):
await websocket_manager.connect(websocket)
try:
while True:
data = await websocket.receive_text()
await websocket_manager.broadcast(f"User {user_id} sent: {data}")
except WebSocketDisconnect:
websocket_manager.disconnect(websocket)
```

Применение этих шаблонов и техник в приложении для управления университетом гарантирует, что асинхронные компоненты приложения будут работать эффективно, хорошо управляться и будут надежными. Ваша система сможет в полной мере использовать асинхронное программирование для обеспечения надежной и оперативной работы с пользователями при условии грамотного подхода к устранению ошибок, эффективного управления ресурсами и модульного построения компонентов.

Отладка асинхронных приложений

Одновременная работа нескольких асинхронных приложений делает их отладку одной из самых сложных задач. Обычные методы отладки могут оказаться неэффективными, если возникают непредсказуемые проблемы или они не соответствуют линейной схеме. Наше программное обеспечение для управления университетом, созданное на основе веб-интерфейса FastAPI, является сложной системой, но с помощью соответствующих инструмен-

тов и методов мы смогли успешно выявить и устранить проблемы с асинхронным кодом.

Инструменты и методы отладки

Пример программы: «Ведение подробного журнала»

В приложениях с асинхронной обработкой данных протоколирование жизненно необходимо, поскольку с его помощью можно получить представление о том, что происходит в приложении в любой момент времени. С помощью протоколирования можно отследить ход выполнения задачи и понять, как обрабатываются данные.

```
import logging
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
import asyncio
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
app = FastAPI()
@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
await websocket.accept()
try:
while True:
data = await websocket.receive_text()
logger.info(f " Полученные данные: {data}")
await websocket.send_text(f "Сообщение получено: {data}")
logger.info(f "Отправлено подтверждение клиенту.")
except WebSocketDisconnect:
logger.warning(f "WebSocket отключен.")
```

При таких настройках журналы позволяют четко отследить момент получения данных и отправки ответов, а также неожиданное отключение клиента.

Пример программы: «Использование сообщений *print* для получения немедленной обратной связи»

Это не самый сложный метод. Операторы печати `print` (или ведение журнала на уровне отладки) для вывода информации о внут-

реннем состоянии или значениях переменных в средах разработки необходимо разместить в стратегически важных местах, что может быть очень полезно для быстрых проверок.

```
@app.post("/enroll/{student_id}/{course_id}")
async def enroll_student(student_id: int, course_id: int):
print(f "Попытка зачислить студента {student_id} на курс
{course_id}")
enrollment = await enroll_student_db(student_id, course_id)
if not enrollment:
print("Зачисление не удалось")
raise HTTPException(status_code=500, detail="Зачисление не удалось")
print("Зачисление прошло успешно")
return { "message": "Студент успешно зачислен"}
```

С помощью этого метода можно получить мгновенную и четкую обратную связь о том, что происходит в функции во время ее выполнения.

Пример программы: «Включение режима отладки *asyncio*»

В модуле `asyncio` предусмотрен режим отладки, который может оказаться очень полезным для выявления таких распространенных проблем, как неожиданные задачи, медленные задачи и неожиданные задачи, которые были отменены. В режиме отладки записывается много дополнительной информации о внутреннем состоянии цикла событий `asyncio`.

```
import os
# Установите переменную окружения перед запуском приложения
os.environ['PYTHONASYNCIODEBUG'] = '1'
# В качестве альтернативы можно программно установить режим отладки
import asyncio
asyncio.get_event_loop().set_debug(True)
```

При отладке асинхронных приложений используются как стандартные методы отладки, так и инструменты, предназначенные для выполнения данного типа задач. Режим отладки в `asyncio`, наряду с подробным протоколированием, встроенными средствами отладки среды разработки (IDE), простыми операторами печати и

другими приемами, позволяет разработчикам эффективно отслеживать и диагностировать ошибки в асинхронных приложениях.

Резюме

В этой главе мы подробно рассмотрели особенности асинхронного программирования на языке Python, сделав акцент на различных деталях, позволяющих программистам создавать эффективные и быстро реагирующие (отзывчивые) приложения. В первой части главы было дано введение в асинхронное программирование с описанием его ключевых особенностей и отличий от обычного синхронного программирования. Далее мы погрузились в основы `asyncio` по написанию программ, работающих в параллельном режиме, с использованием синтаксиса `async/await`. Прочную основу для работы с асинхронными задачами нам заложили практические примеры, в которых было показано, как включить основные асинхронные функции в веб-приложение.

В следующем разделе говорилось о том, как использовать веб-фреймворк FastAPI для создания неблокирующих веб-сервисов, а также рассказывалось о разработке асинхронных веб-приложений. Здесь вы сможете найти примеры реальных приложений, которые управляют веб-сокетами для обеспечения связи в реальном времени и обрабатывают взаимодействие данных в режиме онлайн.

Кроме того, в этом разделе мы уделили внимание асинхронному доступу к базам данных. Нами были рассмотрены методы повышения скорости отклика и производительности приложения под нагрузкой за счет внедрения асинхронно-совместимых драйверов баз данных и выполнения неблокирующих операций с базами данных.

Кроме того, были рассмотрены вопросы работы в параллельном режиме с акцентом на то, как `asyncio` использует этот режим работы для оптимизации времени выполнения асинхронных программ. Также был представлен раздел о том, как создавать веб-сокеты, которые позволяют осуществлять двустороннюю связь между серверами и клиентами в режиме реального времени.

В заключительном разделе главы рассматриваются стратегии отладки, оптимизированные для асинхронных приложений, а также передовой опыт и шаблоны для создания асинхронного кода, который должен быть аккуратным и простым в сопровождении. Для того чтобы сложные асинхронные приложения работали бесперебойно и эффективно, важно знать и владеть методами обработки ошибок, управления ресурсами и отладки, которые рассматриваются в этих разделах.

ГЛАВА 7
Организация работы с пользователями и их безопасность

Введение

В этой главе мы рассмотрим основы безопасности веб-приложений, в частности методы управления пользователями и обеспечение безопасности данных. Глава начинается с раздела о проектировании систем аутентификации пользователей и посвящена тому, как эти системы должны быть структурированы, чтобы правильно работать с личными данными пользователей, поддерживая при этом стандарты безопасности. Здесь вы найдете обзор идей и процессов, поддерживающих аутентификацию пользователей, а также описание необходимых шагов и препятствий, с которыми можете столкнуться.

Следующий раздел главы посвящен реализации протоколов OAuth и JSON Web Tokens (JWT), предназначенных для контроля доступа к ресурсам. Эти протоколы обеспечивают безопасное и эффективное управление правами пользователей и контролем доступа. Протоколы необходимы для обеспечения современной веб-безопасности и широко используются для авторизации в веб-приложениях. Контроль доступа на основе ролей (RBAC) — следующая тема для изучения. Этот способ позволяет ограничить доступ к системе для авторизованных пользователей в соответствии с их ролью в организации. Здесь мы рассмотрим все тонкости разработки и внедрения системы RBAC, что поможет вам точно настроить политику безопасности и предоставить пользователям необходимый доступ.

Еще одна важная тема, рассматриваемая в этой главе, — это обеспечение безопасности REST API. Здесь основное внимание уделяется обеспечению безопасности данных, которыми обмени-

ваются клиенты и серверы, а также стратегиям защиты API от типичных уязвимостей и атак. Кроме того, мы рассмотрим управление пользовательскими сессиями, тактику безопасной работы с ними, например передовой опыт создания, обслуживания и истечения срока действия сессий, который предотвращает перехват и другие риски безопасности.

В целом эти темы дают полное представление о том, как создавать безопасные и надежные системы управления пользователями, защищающие пользователей и данные, к которым они обращаются. Всем, кто работает в сфере разработки или безопасности и хочет сделать свои веб-приложения более безопасными, необходимо ознакомиться с этим материалом.

Проектирование систем аутентификации пользователей

Безопасность нашего университетского приложения, как и любого другого, зависит от того, насколько хорошо мы разработаем систему аутентификации пользователей. Подобная система проверяет учетные данные каждого, кто пытается воспользоваться платформой, чтобы убедиться, что он тот, за кого себя выдает. В этом разделе мы рассмотрим основы построения надежной системы аутентификации пользователей для университетского приложения, а также все необходимые шаги и моменты, о которых следует помнить.

Определение требований

Первым шагом в разработке системы аутентификации является определение специфических требований приложения. Для университетского приложения рассмотрим следующие задачи.

- В приложении, скорее всего, должны поддерживаться различные типы пользователей, например студенты, преподаватели и административный персонал, каждый из которых имеет свои права доступа.
- Конфиденциальная информация, например оценки студентов или данные о зарплате преподавателей, требует более высоких

мер обеспечения безопасности по сравнению с общей информацией, такой как описание курсов.

- Система должна быть простой в использовании, чтобы все пользователи могли успешно взаимодействовать с ней, не сталкиваясь с препятствиями.
- По мере развития университета система должна быть способна обрабатывать все большее количество пользователей и сеансов без снижения производительности.
- В зависимости от местонахождения университета могут действовать разные нормативные требования, касающиеся обработки и защиты данных студентов (например, FERPA в США).

Компоненты системы аутентификации пользователей

Учитывая вышеописанные требования, вы можете приступить к проектированию системы, определив необходимые компоненты.

- *Схема базы данных.* Разработайте структуру базы данных для безопасного хранения учетных данных пользователей. Обычно сюда входит таблица `users` с полями для идентификатора пользователя, имени пользователя, пароля (хешированного), электронной почты и, возможно, идентификатора роли.
- *Процесс регистрации.* Предоставьте новым пользователям возможность зарегистрироваться. Обычно этот процесс включает сбор необходимой информации, проверку адреса электронной почты или номера мобильного телефона, а также безопасную обработку паролей.
- *Процесс входа в систему.* Создайте безопасный механизм входа в систему. В него должна быть включена форма для ввода имени пользователя и пароля, которые должны передаваться по защищенному соединению (HTTPS).
- *Управление сеансом.* После аутентификации система должна создать сессию для пользователя, которая сохраняется на сайте, чтобы узнавать этого пользователя при последующих запросах без необходимости повторной аутентификации.

- *Хранение паролей.* Используйте безопасные методы хранения паролей, например для защиты паролей в вашей базе данных примените алгоритм хеширования с солью.

> ***Примечание переводчика***
>
> Хеширование с солью — это процесс, в котором к входным данным для вычисления хеша присоединяется дополнительная строка (соль). За счет дополнительной строки получается совершенно другое хеш-значение, которое невозможно найти в стандартных таблицах.

Пример программы: «Подключение аутентификации пользователей»

Ниже приводится упрощенная схема реализации системы аутентификации пользователей в веб-фреймворке Flask для Python:

1. Создайте приложение на основе фреймворка Flask.

```
from flask import Flask, request, session
from flask_bcrypt import Bcrypt
app = Flask(__name__)
app.secret_key = 'your_secret_key_here'
bcrypt = Bcrypt(app)
```

2. Конечная точка регистрации пользователей:

```
@app.route('/register', methods=['POST'])
def register():
username = request.form['username']
password = request.form['password']
hashed_password =
          bcrypt.generate_password_hash(password).decode('utf-8')
# Сохраните имя пользователя и хешированный_пароль в базе данных
return "Пользователь успешно зарегистрирован"
```

3. Конечная точка входа пользователя в систему:

```
@app.route('/login', methods=['POST'])
def login():
username = request.form['username']
password = request.form['password']
```

```
# Получение хешированного пароля пользователя из базы данных
user_hashed_password = 'fetched_from_db'
if bcrypt.check_password_hash(user_hashed_password, password):
session['user'] = username
return "Вход в систему выполнен успешно"
else:
return "Неверные учетные данные", 401
```

При разработке каждого из этих компонентов учитывайте особенности университетского приложения с точки зрения удобства использования и безопасности, а также убедитесь, что приложение может масштабироваться в соответствии с нормативными требованиями.

Реализация стандартов OAuth и JWT

В наши дни ни одно решение в области веб-безопасности не обходится без применения стандартов OAuth и JSON Web Tokens, которые становятся незаменимыми, когда аутентификация и авторизация пользователей в нескольких системах требуют дополнительного уровня безопасности. Давайте рассмотрим эти технологии более внимательно и узнаем, как мы можем включить их во внутреннюю часть нашего университетского приложения.

Введение в OAuth

OAuth — это открытый стандарт передачи прав доступа. Этот стандарт обычно используется для предоставления интернет-пользователям доступа к информации на других сайтах, но без передачи паролей. Стандарт разработан для работы по протоколу HTTP и предоставляет токены вместо учетных данных для доступа к своим данным, размещенным у определенного поставщика услуг. OAuth особенно полезен при попытке обеспечить безопасный и эффективный способ входа пользователей в систему с помощью внешних сервисов (например, Google, Facebook или Twitter) без раскрытия пароля.

Введение в веб-токены JSON (JWT)

Веб-токен JWT — это компактное, безопасное для адреса URL средство представления требований, которые должны быть переданы между двумя сторонами. Утверждения в JWT закодированы в виде объекта JSON, который используется как полезная нагрузка структуры JSON Web Signature (JWS) или как открытый текст структуры JSON Web Encryption (JWE). С помощью этого средства можно поставить цифровую подпись или защитить целостность с помощью кода аутентификации сообщения (MAC) и/или зашифровать утверждения.

Токены JWT используются в протоколах аутентификации и авторизации, включая стандарт OAuth 2.0, поскольку они основаны на формате JSON, более компактны и понятны, чем токены SAML на базе XML.

Пример программы: «Реализация OAuth и JWT»

Интеграция стандартов OAuth и JWT в наше университетское приложение может способствовать безопасной и масштабируемой аутентификации и авторизации пользователей, особенно при доступе и взаимодействии со сторонними сервисами.

Настройка OAuth с помощью провайдера

Предположим, вы хотите разрешить пользователям входить в систему, используя их учетные записи Google. Вы начнете с настройки учетных данных OAuth в Google:

1. Зайдите в Google Cloud Console.
2. Создайте новый проект или выберите существующий.
3. Перейдите в раздел **API и службы** | **Учетные данные**.
4. Щелкните мышью на ссылке **Создать учетные данные** и выберите **Идентификатор клиента OAuth**.
5. Настройте экран согласия и выберите тип приложения — веб-приложение.

Примечание переводчика

Экран согласия — это специальный экран, на котором пользователь дает согласие на сбор, обработку и использование своих персональных данных.

6. Добавьте авторизованные URI перенаправления, которые будут обрабатывать ответ от Google.

При создании учетных данных вы получите идентификатор клиента и секрет клиента. Храните их в надежном месте.

Интеграция OAuth в приложение

Если вы используете Flask, воспользуйтесь библиотекой типа **Authlib** или **Flask-OAuthlib**:

```
from authlib.integrations.flask_client import OAuth
app = Flask(__name__)
oauth = OAuth(app)
google = oauth.register(
name='google',
client_id='YOUR_GOOGLE_CLIENT_ID',
client_secret='YOUR_GOOGLE_CLIENT_SECRET',
access_token_url='https://accounts.google.com/o/oauth2/token',
access_token_params=None,
authorize_url='https://accounts.google.com/o/oauth2/auth',
authorize_params=None,
api_base_url='https://www.googleapis.com/oauth2/v1/',
client_kwargs={'scope': 'openid profile email'},
)
```

Реализация JWT для управления сеансами

После аутентификации пользователя с помощью стандарта OAuth необходимо сгенерировать токен стандарта JWT, предназначенный для безопасного управления сессией:

```
import jwt
from datetime import datetime, timedelta
def create_token(user_id):
payload = {
'exp': datetime.utcnow() +timedelta(days=1), # Срок действия токена
                                             # истекает через один день
```

```
'iat': datetime.utcnow(), # Токен выписан на
'sub': user_id            # субъект (к которому относится токен)
}
return jwt.encode(payload, 'YOUR_SECRET_KEY', algorithm='HS256')
@app.route('/login/callback')
def login_callback():
token = google.authorize_access_token()
resp = google.get('userinfo')
user_info = resp.json()
# Создание JWT-токена после успешной аутентификации
# по стандарту OAuth
jwt_token = create_token(user_info['id'])
return {'jwt_token': jwt_token}
```

Благодаря стандарту OAuth студенты для доступа к университетскому приложению могут использовать свои существующие учетные записи Google, что упрощает процедуру входа и повышает безопасность (пароли не хранятся непосредственно в приложении). Дополнительную защиту приложения обеспечивает использование стандарта JWT, предназначенного для обработки клиент-серверных сессий и подтверждений после входа в систему. Это обеспечивает эффективное и безопасное управление пользовательскими сессиями.

Контроль доступа на основе ролей (RBAC)

RBAC (Role-Based Access Control или управление доступом на основе ролей) — это способ, с помощью которого предприятия могут контролировать, у кого из сотрудников какой доступ к сетям и компьютерам в зависимости от должности. Применение метода RBAC крайне важно для работы университетского приложения, поскольку он гарантирует, что пользователи (студенты, преподаватели и администраторы) смогут получить доступ только к тем данным и функциям, которые имеют отношение к их работе.

Определение эффективности RBAC

Метод RBAC эффективен, поскольку упрощает управление разрешениями пользователей. Вместо того чтобы назначать разре-

шения каждому пользователю отдельно, для каждого набора разрешений создаются роли, и эти роли назначаются пользователям. Это упрощает управление сложными разрешениями и обеспечивает последовательное применение политики безопасности во всем приложении.

Пример программы: «Внедрение RBAC»

Чтобы продемонстрировать реализацию метода RBAC, рассмотрим веб-приложение, написанное на языке Python и использующее фреймворк Flask, поскольку этот фреймворк часто используется для создания веб-приложений.

Определение ролей и разрешений

Сначала определите, какие роли уже существуют и какими правами каждая роль должна обладать. Для университетского приложения вы можете определить такие роли, как:

- `Student` — студент может просматривать расписание занятий, оценки и записываться на курсы;
- `Professor` — профессор может просматривать и управлять своими курсами, оценивать задания;
- `Admin` — администратор может управлять учетными записями пользователей, редактировать информацию обо всех курсах, просматривать все записи студентов.

Разрешения могут выглядеть следующим образом:

- `view_grades` — просмотр оценок;
- `edit_courses` — редактирование курсов;
- `view_courses` — просмотр курсов;
- `manage_users` — управление пользователями.

Настройка среды

1. Установите фреймворк Flask и библиотеку Flask-SQLAlchemy для работы с приложением и базой данных.

```
pip install Flask Flask-SQLAlchemy
```

2. Настройте приложение и базу данных.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI']= 'sqlite:///university.db'
db = SQLAlchemy(app)
```

Определение моделей пользователей и ролей

Определите в библиотеке SQLAlchemy модели для `User`, `Role` и создайте вспомогательную таблицу для отношений «многие-ко-многим» между пользователями и ролями.

```
class User(db.Model):
id = db.Column(db.Integer, primary_key=True)
username = db.Column(db.String(80), unique=True, nullable=False)
roles = db.relationship('Role', secondary='user_roles')
class Role(db.Model):
id = db.Column(db.Integer, primary_key=True)
name = db.Column(db.String(50), unique=True, nullable=False)
class UserRoles(db.Model):
user_id = db.Column(db.Integer, db.ForeignKey('user.id'),
primary_key=True)
role_id = db.Column(db.Integer, db.ForeignKey('role.id'),
primary_key=True)
```

Создание ролей

Инициализируйте базу данных и добавьте роли.

```
with app.app_context():
db.create_all()
student_role = Role(name="Student")
professor_role = Role(name="Professor")
admin_role = Role(name="Admin")
db.session.add_all([student_role, professor_role, admin_role])
db.session.commit()
```

Назначение ролей пользователям

При создании нового пользователя назначьте ему соответствующие роли.

```
new_user = User(username="john_doe")
new_user.roles.append(student_role)
db.session.add(new_user)
db.session.commit()
```

Обеспечение проверки ролей

Создайте простую проверку роли для маршрутов.

```
from flask import abort
def role_required(*roles):
def wrapper(fn):
@wraps(fn)
def decorated_view(*args, **kwargs):
if not current_user or not any(role.name in roles for role
in current_user.roles):
abort(403) # Запрещенный доступ
return fn(*args, **kwargs)
return decorated_view
return wrapper
@app.route('/view_grades')
@role_required('Student', 'Professor')
def view_grades():
return "Grades"
```

В приведенном выше примере программы мы видим, как создаются роли пользователей, а затем используются для реализации контроля доступа. Этот метод упрощает работу программы с разрешениями пользователей и позволяет убедиться, что только авторизованные пользователи могут выполнять действия, что и обеспечивает безопасность и конфиденциальность данных.

Обеспечение безопасности REST API

Что необходимо для обеспечения безопасности REST API?

REST API часто выступают в качестве интерфейса для передачи данных и обеспечения функциональности веб-сервера, что делает их главной мишенью для атак. Без надлежащей защиты API мо-

гут раскрывать конфиденциальные данные, предоставлять несанкционированный доступ и потенциально приводить к сбоям в работе сервиса. Защита REST API необходима для того, чтобы данные клиента и сервера оставались конфиденциальными, неприкосновенными и доступными только для авторизованных пользователей. В данном методе используется ряд технологий для защиты API от типичных направлений атак и устранения уязвимостей, включая атаки типа «человек посередине» (MitM), SQL-инъекции, межсайтовый скриптинг (XSS) и другие.

Стратегии обеспечения безопасности REST API

Использование HTTPS

Первый и самый простой шаг — обеспечить шифрование всех соединений между клиентом и сервером с помощью протокола HTTPS. Это предотвращает перехват и чтение ваших данных в ходе транспортировки злоумышленниками.

```
# Убедитесь, что приложение Flask использует протокол HTTPS
from flask import Flask, request, jsonify, redirect
app = Flask(__name__)
@app.before_request
def before_request():
if not request.is_secure:
url = request.url.replace('http://', 'https://', 1)
code = 301
return redirect(url, code=code)
```

Аутентификация и авторизация

И в процессе аутентификации, и в процессе авторизации проверяется легитимность пользователей перед тем, как предоставить им доступ к ресурсам. Для безопасного и эффективного управления сеансами и аутентификации используйте стандарт JWT, как показано ниже:

```
import jwt
from flask import request, abort
@app.route('/api/resource')
```

```
def get_resource():
token = request.headers.get('Authorization')
try:
payload = jwt.decode(token, "SECRET_KEY", algorithms=["HS256"])
except (jwt.DecodeError, jwt.ExpiredSignatureError):
abort(401) # Несанкционированный доступ
user_id = payload['sub']
# Проверяем, имеет ли user_id доступ к этому ресурсу
return jsonify({"data": "Secure data"})
```

Валидация и дезинфекция входных данных

Чтобы защитить свой API от SQL-инъекций и других форм инъекционных атак, всегда проверяйте и обеззараживайте вводимые пользователем данные:

```
from flask import request
from marshmallow import Schema, fields, ValidationError
class ResourceSchema(Schema):
resource_id = fields.Int(required=True)
@app.route('/api/resource', methods=["POST"])
def create_resource():
json_input = request.get_json()
try:
data = ResourceSchema().load(json_input)
except ValidationError as err:
return jsonify(err.messages), 400
resource_id = data['resource_id']
# Безопасная обработка действительного resource_id
return jsonify({"message": " Ресурс создан "}), 201
```

Примечание переводчика

SQL-инъекция (или «внедрение SQL-кода») — это метод получения несанкционированного доступа к базе данных, при котором вредоносный код выполняется прямо из поля ввода в обычной форме. Если SQL-инъекция прошла успешно, неавторизованные лица могут читать, создавать, обновлять или даже удалять записи из таблиц базы данных.

Внедрение ограничения скорости

Ограничение скорости позволяет предотвратить злоупотребления и DoS-атаки (отказ в обслуживании), ограничивая количество запросов, которые может сделать пользователь или IP-адрес за определенный промежуток времени.

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
limiter = Limiter(
app,
key_func=get_remote_address,
default_limits=["200 per day", "50 per hour"]
)
@app.route("/api/resource")
@limiter.limit("10 per minute")
def get_resource():
return jsonify({"data": " Скорость ограничена "})
```

Приведенные выше тактики предоставляют надежную стратегию защиты REST API от различных рисков безопасности, что, в свою очередь, защищает ваши данные и ваших пользователей.

Управление сеансами пользователей

Для обеспечения безопасности онлайн-приложений необходимо использовать управление пользовательскими сеансами. Это особенно важно при работе с протоколами HTTP без учета статистики и взаимодействия с пользователями. С момента входа пользователя в систему и до момента выхода из нее процедуры безопасного управления сеансами защищают пользовательские данные, предотвращая несанкционированный доступ и перехват сеанса.

Основные понятия

- *Создание сеанса.* При успешном входе пользователя в систему сервер создает сессию, которая обычно представлена идентификатором сессии (ID сессии). Этот идентификатор сессии должен быть уникальным и не поддаваться прогнозированию,

чтобы потенциальные злоумышленники не смогли угадать или подделать действительный идентификатор сессии.

- *Хранение данных сеанса.* Данные сессии могут храниться как на стороне клиента (в cookies), так и на стороне сервера (в базе данных или в памяти). Каждый подход имеет свои последствия для производительности и безопасности.
- *Срок действия сессии.* У каждой сессии должно быть определенное время жизни, по истечении которого сессия автоматически закрывается. Это предотвращает неограниченное использование старых сессий, что может представлять угрозу безопасности.
- *Обслуживание сеансов.* Во время сеанса важно по мере необходимости контролировать и обновлять данные сеанса, обеспечивая при этом его безопасность от таких угроз, как фиксация или перехват.

Реализация безопасного управления сеансами

Используя веб-фреймворк Flask, вы можете реализовать все вышеперечисленные особенности управления сессиями.

Создание и обработка сеансов

Когда пользователь входит в систему, фреймворк Flask автоматически создает сессию. Для предотвращения несанкционированного доступа механизм сессий Flask подписывает файлы cookie:

```
from flask import Flask, session, request, redirect, url_for
app = Flask(__name__)
app.secret_key = 'your_secret_key' # Секретный ключ для подписания
                                   # cookie сеанса
@app.route('/login', methods=['POST'])
def login():
username = request.form['username']
password = request.form['password']
user = validate_user(username, password)
if user:
session['user_id'] = user.id # Хранит идентификатор пользователя
                             # в сессии
```

```
return redirect(url_for('dashboard'))
return 'Invalid username or password', 401
```

Сохранение сеансов

По умолчанию сессии во Flask хранятся в cookies на стороне клиента. Для большей безопасности с помощью таких расширений, как Flask-Session, вы можете настроить Flask на хранение данных сессии на стороне сервера:

```
from flask_session import Session
app.config['SESSION_TYPE'] = 'filesystem' # Хранит сессии
                                          # в файловой системе
Session(app)
```

Такая конфигурация повышает безопасность за счет хранения фактических данных сеанса на стороне сервера и отправки клиенту только идентификатора сеанса, что снижает риск несанкционированного вмешательства со стороны клиента.

Истечение срока действия сеанса

Сеансы должны быть настроены так, чтобы они заканчивались после определенного периода бездействия или через фиксированный промежуток времени:

```
from datetime import timedelta
app.config['PERMANENT_SESSION_LIFETIME'] = timedelta(hours=1) # Срок
                                  действия сессии истекает через 1 час
```

Этот параметр гарантирует, что сеансы пользователей не останутся активными на неопределенный срок. Таким образом, можно предотвратить риск перехвата сеанса, особенно на общедоступных или совместно используемых компьютерах.

Обслуживание и безопасность сеансов

Во время сеанса пользователя определенные методы помогают поддерживать безопасность:

- Для предотвращения атак с фиксацией сеанса восстановите идентификатор сеанса при аутентификации или при смене роли пользователя.

```
@app.route('/login', methods=['POST'])
def login():
# Предполагаем, что пользователь подтвердил свою личность
session.regenerate()  # Регенерировать идентификатор сессии
return 'Вход в систему выполнен успешно'
```

- Убедитесь, что все сессионные файлы cookie отправляются по протоколу HTTPS, чтобы защитить их от перехвата злоумышленниками.

```
app.config['SESSION_COOKIE_SECURE'] = True
```

- Настройте сессионные cookies на `HttpOnly` и установите атрибут `SameSite`, чтобы предотвратить межсайтовый скриптинг (XSS) и подделку межсайтовых запросов (CSRF).

```
app.config['SESSION_COOKIE_HTTPONLY'] = True
app.config['SESSION_COOKIE_SAMESITE'] = 'Lax'
```

> ***Примечание переводчика***
>
> HttpOnly Cookie — это особый тип cookie, который доступен только на серверной части сайта или приложения. Он используется для хранения данных в форме пары ключ/значение, и с этим типом данных в основном работает сервер. Основное отличие HttpOnly Cookie от обычных — к ним не имеет доступа клиентское приложение (браузер).

Данные пользователя и целостность приложения защищены, если реализованы процедуры безопасного управления сеансами. Эти методы помогают предотвратить такие типичные уязвимости, как перехват и фиксация сеанса.

Внедрение двухфакторной аутентификации

Описание двухфакторной аутентификации

Двухфакторная аутентификация, или 2FA — это метод защиты, при котором для подтверждения подлинности пользователя запрашиваются два отдельных фрагмента информации, что значительно повышает степень защиты вашего приложения. Этот метод обеспечивает защиту учетных записей пользователей

в случае компрометации одного фактора, например пароля. Поскольку в системе университетских приложений существует множество различных типов пользователей и ролей, двухфакторная аутентификация (2FA) является хорошим способом защиты важной информации.

Двухфакторная аутентификация обычно сочетает в себе информацию, которую знает пользователь (например, пароль), с информацией, которая доступна пользователю (например, приложение для смартфона, генерирующее код, основанный на времени), или с информацией, которая есть у пользователя (например, отпечаток пальца). Для университетского приложения мы остановимся на общепринятом подходе, включающем пароль и одноразовый пароль, основанный на времени (TOTP), генерируемый приложением вроде Google Authenticator или Authy.

Шаги по внедрению 2FA

Генерация секретного ключа для пользователя

Если пользователь решил использовать двухфакторную авторизацию, сгенерируйте секретный ключ, который будет использоваться для генерации TOTP. Этот ключ должен надежно храниться в профиле пользователя в базе данных.

```
import pyotp
def generate_secret():
return pyotp.random_base32()
# Допустим, у нас есть функция для сохранения этих данных
# в профиле пользователя
def setup_2fa(user_id):
secret = generate_secret()
save_secret_to_user_profile(user_id, secret)
return secret
```

> ***Примечание переводчика***
>
> Time-based One-time Password (TOTP) или одноразовый пароль, основанный на времени, — это алгоритм создания одноразовых паролей для защищенной аутентификации. Является улучшением алгоритма HOTP (HMAC-Based One-time Password Algorithm или алгоритм одноразового ввода пароля на основе HMAC).

Принцип работы: клиент берет текущее значение таймера и секретный ключ, хеширует их с помощью какой-либо хеш-функции и отправляет серверу. В свою очередь сервер проводит те же вычисления, после чего ему остается только сравнить эти значения.

Главное отличие TOTP от HOTP — это генерация пароля на основе времени, т. е. время является параметром. При этом обычно используется не точное указание времени, а текущий интервал с установленными заранее границами (обычно 30 секунд).

Связывание секретного ключа с приложением Authenticator

Секретный ключ необходимо ввести в приложение, генерирующее TOTP. Часто для этого пользователю показывают QR-код, который он может отсканировать с помощью приложения.

```
import pyotp
def get_totp_uri(secret, user_email):
totp = pyotp.TOTP(secret)
return totp.provisioning_uri(user_email, issuer_name="UniversityApp")
```

Примечание переводчика

Google Authenticator — это приложение для двухфакторной (или двухэтапной) аутентификации, которое генерирует одноразовые коды подтверждения для более надежной защиты аккаунта.

Проверка TOTP во время входа в систему

Когда пользователь войдет в систему под своим именем и паролем, предложите ему ввести код из приложения TOTP. Затем проверьте этот код, используя секрет, хранящийся в профиле пользователя.

```
def verify_totp(token, user_id):
user_secret = get_secret_from_user_profile(user_id)
totp = pyotp.TOTP(user_secret)
return totp.verify(token)
```

Интеграция 2FA в процесс входа в систему

Измените процесс входа, чтобы после успешной проверки первого пароля включить второй шаг для проверки 2FA.

```
from flask import request, session
@app.route('/login', methods=['POST'])
def login():
username = request.form['username']
password = request.form['password']
if authenticate(username, password):
# Пользователю предлагается ввести токен 2FA после проверки пароля
session['pre_2fa_auth_user_id'] = get_user_id(username)
return 'Enter 2FA code'
return 'Login failed', 401
@app.route('/verify_2fa', methods=['POST'])
def verify_2fa():
token = request.form['token']
user_id = session.pop('pre_2fa_auth_user_id', None)
if user_id and verify_totp(token, user_id):
# Успешная верификация 2FA
session['user_id'] = user_id
return 'Login successful'
return 'Invalid 2FA token', 401
```

В этот процесс входят безопасная генерация секретного ключа, привязка пользовательского ключа к приложению, генерирующему TOTP, и проверка кода TOTP при входе в систему. Такая интеграция значительно повышает безопасность приложения, но требует осторожного обращения с криптографическими компонентами и пользовательскими данными.

Резюме

В этой главе мы подробно рассмотрели основы управления учетными данными и безопасности веб-приложений, уделив особое внимание наиболее важным моментам в обеспечении безопасности пользовательской информации и реализации надежных средств управления доступом. Начав с обзора фундаментальных требований по эффективному определению и проверке учетных данных пользователей, мы перешли к процессу проектирования систем аутентификации пользователей. В этом разделе мы более подробно рассмотрели современные методы аутентификации, такие как стандарты OAuth и JWT, которые необходимы для

обеспечения безопасного доступа к ресурсам без раскрытия паролей пользователей, и заложили основу для их дальнейшего практического применения.

Далее мы познакомились с методом RBAC — важнейшим подходом к ограничению доступа к системе в соответствии с ролями пользователей. Кроме того, мы уделили особое внимание безопасности REST API, сосредоточившись на защите конечных точек от распространенных угроз безопасности с помощью таких мер, как проверка ввода, аутентификация на основе маркеров и HTTPS. Также в этой главе рассматриваются безопасные способы хранения и защиты паролей с акцентом на использование надежных алгоритмов хеширования и засолки. Для защиты от перехвата сеанса и других уязвимостей было рассмотрено управление сеансами пользователей и методы безопасной обработки сеансов пользователей от создания до истечения срока действия сеанса.

Наконец, была предложена идея внедрения двухфакторной аутентификации как способа повысить уровень безопасности, добавив еще один кордон предотвращения несанкционированного доступа. Перечисленные в этой главе шаги представляют собой всеобъемлющую стратегию управления пользователями и обеспечения безопасности.

ГЛАВА 8
Развертывание внутренних приложений, написанных на языке Python

Введение

В этой главе мы рассмотрим основные принципы развертывания внутренних приложений, написанных на языке Python. Здесь будет рассказано об основных инструментах и методах, позволяющих обеспечить масштабируемость, безопасность и производительность ваших веб-приложений. Эта глава начинается с краткого введения в программную платформу Docker и методик создания контейнеров, а затем мы узнаем, как с помощью этих технологий можно создавать изолированные среды приложений.

Далее мы рассмотрим, как использовать Docker для приложений, написанных на языке Python, в частности покажем, как можно выполнять их контейнерную обработку для более удобного развертывания и перемещения. В следующем разделе мы познакомимся с Kubernetes, платформой для организации контейнеров. Эта платформа помогает приложениям справляться с повышенными нагрузками за счет автоматического масштабирования и управления множеством контейнеров в кластерах серверов.

В продолжение темы интеграции и автоматизации, изложенной в предыдущей главе, здесь мы рассмотрим применение передового опыта CI/CD ко внутренним приложениям, которые написаны на языке Python. Кроме того, мы рассмотрим функцию Nginx как обратного прокси, т. е. посредника между клиентскими запросами. Благодаря этому можно решить такие задачи, как завершение создания SSL, балансировка нагрузки и повышение безопасности приложений, написанных на языке Python. Далее мы рассмотрим

SSL-сертификаты и настройки протокола HTTPS, которые крайне важны для защиты конфиденциальных данных и сохранения доверия конечных пользователей, а также способы защиты данных приложения во время их передачи.

В последнем разделе главы речь пойдет о способах масштабирования приложений Python, которые дают возможность приложению быстро и эффективно обрабатывать запросы пользователей.

Обзор Docker и контейнеров

Будучи простой заменой традиционной виртуализации, технология контейнеризации внесла существенные изменения в циклы развертывания и управления приложениями. Docker — наиболее известная и широко используемая платформа среди доступных в настоящее время инструментов контейнеризации. С помощью Docker программисты могут создавать стандартизированные блоки для разрабатываемого программного обеспечения, называемые контейнерами, которые содержат приложение и все его зависимости.

Описание контейнерной технологии

Контейнеризация подразумевает инкапсуляцию приложения и его окружения — зависимостей, конфигураций, скриптов и двоичных файлов — в один контейнер. В отличие от виртуальных машин (ВМ), которым требуется собственная операционная система, контейнеры используют общее ядро операционной хост-системы, но поддерживают изолированные процессы, файловые системы и сетевое взаимодействие. Это делает контейнеры гораздо более эффективными, более простыми в управлении и менее ресурсоемкими, чем традиционные виртуальные машины.

Роль Docker в контейнеризации

Docker предоставляет инструменты и платформу для управления жизненным циклом контейнеров. Он стал синонимом контейнерной технологии благодаря простоте использования, обширному инструментарию и надежной экосистеме. Docker использует про-

стой синтаксис, который позволяет создавать, отправлять и запускать контейнеры с помощью ряда простых команд.

Ниже перечислены ключевые компоненты докера.

- *Dockerfile.* Это текстовый документ, содержащий все команды, которые пользователь может вызвать в командной строке для сборки образа. С помощью **docker build** пользователи могут запустить автоматизированную сборку, которая последовательно выполняет несколько команд командной строки.
- *Образы Docker.* Это неизменяемый файл, который, по сути, является моментальным снимком контейнера. Образы служат строительными блоками Docker. Они могут храниться в реестре и использоваться для создания контейнеров.
- *Контейнеры Docker.* Экземпляр образа Docker. Контейнер служит для выполнения одного приложения, процесса или службы.
- *Хаб/реестр Docker.* Это сервис, предоставляющий пользователям Docker место для размещения и обмена образами. На общедоступном хабе Docker Hub, поддерживаемом компанией Docker Inc., размещаются тысячи образов с предварительно упакованными приложениями, сервисами и утилитами.

Доминирующее положение Docker

Доминирование Docker можно объяснить несколькими факторами.

- *Возможность переноса.* После создания контейнера с помощью Docker этот контейнер можно запустить на любой машине с установленной программной платформой Docker, независимо от базовой среды. Это устраняет проблему «на моей машине работает».
- *Контроль версий и возможность повторного использования.* Образы могут быть разделены на версии, данные о версиях храниться в реестре, а сами версии могут использоваться многократно. Организации способны повысить свою производительность за счет использования базовых образов, которые

поддерживаются и обновляются с помощью патчей безопасности.

- *Изоляция.* Docker гарантирует, что приложения, запущенные в контейнерах, изолированы не только друг от друга, но и от хост-системы. Такая изоляция способствует безопасности и позволяет нескольким контейнерам работать на одном хосте без каких-либо помех.
- *Масштабируемость и микросервисы.* Легковесность программной платформы Docker позволяет легко масштабировать систему. Контейнеры можно быстро запускать и останавливать, что позволяет более эффективно справляться со скачками спроса на приложения. Docker благоприятствует архитектуре микросервисов, поскольку позволяет размещать каждую часть приложения в отдельных контейнерах и масштабировать ее независимо.

Пример программы: «Использование Docker»

Чтобы проиллюстрировать, как правильно использовать Docker, давайте рассмотрим простой пример контейнеризации приложения, написанного на языке Python с использованием фреймворка Flask, как показано ниже:

1. Создайте файл Dockerfile:

```
# Используйте официальную среду выполнения Python
# в качестве родительского образа
FROM python:3.8-slim
# Установите рабочий каталог в контейнере
WORKDIR /app
# Скопируйте содержимое текущего каталога
# в контейнер по адресу /app
ADD . /app
# Установите все необходимые пакеты, указанные
# в файле requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt
# Сделайте порт 80 доступным для окружающего мира
# за пределами этого контейнера
EXPOSE 80
```

```
# Определите переменную окружения
ENV NAME World
# Запускаем app.py при запуске контейнера
CMD ["python", "app.py"]
```

2. Сборка образа Docker.

```
docker build -t my-python-app
```

3. Запуск контейнера Docker.

```
docker run -p 4000:80 my-python-app
```

В этой установке в файле Dockerfile задается окружение и описываются шаги для запуска приложения Python Flask. Образ собирается с учетом этих спецификаций, а затем из этого образа запускается контейнер.

Использование Docker для приложений, написанных на языке Python

Приложения, написанные на языке Python, можно с помощью программной платформы Docker легко упаковать в контейнер. Эта процедура упрощает развертывание, гарантирует согласованность в разных средах и особенно полезна для управления зависимостями и настройками. Сейчас я покажу вам, как упаковать в контейнер университетское приложение, над которым мы работали, и как установить Docker в нашу существующую среду разработки.

Установка Docker

Перед началом упаковки приложения в контейнер в системе Linux необходимо установить Docker.

Для этого выполните следующие шаги:

1. Обновите индекс пакетов:

```
sudo apt-get update
```

2. Установите пакеты, позволяющие программе Apt использовать репозиторий по протоколу HTTPS:

```
sudo apt-get install \
apt-transport-https \
ca-certificates \
curl \
gnupg \
lsb-release
```

> ***Примечание переводчика***
>
> Apt (Advanced Packaging Tool или усовершенствованный инструмент для упаковки) — программа для установки, обновления и удаления программных пакетов в операционных системах Debian и основанных на них дистрибутивах (например: Ubuntu, Linux Mint и т. п.).

3. Добавьте официальный GPG-ключ Docker:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
--dearmor -o /usr/share/keyrings/docker-archivekeyring.gpg
```

4. Настройте стабильный репозиторий:

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/dockerarchive- keyring.gpg]
https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable" | sudo
tee/etc/apt/sources.list.d/docker.list > /dev/null
```

5. Установите движок Docker Engine:

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

6. Убедитесь, что Docker установлен правильно, запустив образ `hello-world`:

```
sudo docker run hello-world
```

Приведенные выше действия по установке и настройке гарантируют, что программная платформа Docker установлена и работает в вашей системе. В результате запуска контейнера `hello-world` на экране появится обычный тест для проверки того, что Docker может извлекать и запускать образы.

Контейнеризация университетского приложения

Теперь выполним упаковку университетского приложения в контейнер, произведя описанные далее действия.

Подготовка приложения

Убедитесь, что приложение, созданное на основе Flask, содержит все необходимые файлы. Например, у вас должны быть следующие файлы:

- **app.py** — файл приложения, созданного на основе фреймворка Flask;
- **requirements.txt** — файл со списком всех зависимостей.

Ниже приведен простой пример модуля **app.py** для демонстрации этого процесса:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
return 'Hello, World!'
if __name__ == '__main__':
app.run(host='0.0.0.0')
```

Создание Dockerfile

В файле Dockerfile содержатся инструкции по сборке образа.

```
# Используйте официальную среду выполнения Python
# в качестве родительского образа
FROM python:3.8-slim
# Установите рабочий каталог по адресу /app
WORKDIR /app
# Скопируйте содержимое текущего каталога в контейнер по адресу /app
COPY . /app
# Установите все необходимые пакеты, указанные
# в файле requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
# Сделайте порт 5000 доступным для внешнего мира
# за пределами этого контейнера
EXPOSE 5000
```

```
# Определите переменную окружения
ENV NAME World
# Настройте запуск файла app.py при запуске контейнера
CMD ["python", "app.py"]
```

Сборка образа Docker

Перейдите в каталог, содержащий ваш Dockerfile, и запустите его:

```
docker build -t university-app
```

Эта команда создает образ Docker с тегом `university-app`.

Запуск контейнера Docker

Чтобы запустить экземпляр вашего образа Docker, выполните команду:

```
docker run -p 4000:5000 university-app
```

Эта команда сопоставляет порт 5000 контейнера с портом 4000 на вашей операционной системе, позволяя вам получить доступ к приложению на основе Flask через **localhost:4000** в вашем браузере. Теперь это университетское приложение доступно в контейнерной версии, в которой аккуратно упакованы все зависимости. Вы можете развернуть его в любой среде, поддерживающей Docker, что упрощает развертывание и делает ваше приложение более масштабируемым и надежным.

Kubernetes для управления приложениями

Kubernetes (или сокращенно K8s) — это бесплатная платформа с открытым исходным кодом, предназначенная для управления и автоматизации процесса развертывания и масштабирования контейнерных приложений. Эта платформа упорядочивает контейнеры приложения в логические единицы для более удобного обнаружения и управления. Kubernetes отлично справляется с организацией сложных приложений, особенно тех, которые должны быть развернуты в нескольких средах, масштабироваться и обладать высокой доступностью.

Описание Kubernetes

Kubernetes — это платформа для планирования и запуска контейнеров на кластерах физических или виртуальных машин. Она позволяет отказаться от базовой инфраструктуры, делая развертывание приложений беспрепятственным и масштабируемым.

Основные элементы Kubernetes.

- Капсулы — это наименьшие разворачиваемые единицы, создаваемые и управляемые Kubernetes, обычно состоящие из одного или нескольких контейнеров, которые совместно используют ресурсы хранения и сети.
- Узлы — в зависимости от кластера узел может представлять собой виртуальную или физическую машину. На каждом узле размещается несколько капсул.
- Сервисы — абстракция, определяющая логический набор подсистем и политику доступа к ним, может включать механизм обнаружения сервисов, управляемых Kubernetes.
- Развертывание — управляет развертыванием и масштабированием набора подсистем, а также обеспечивает обновление контейнеров приложения.

Установка и настройка Kubernetes

Для целей разработки, особенно для нашего университетского приложения, **Minikube** является важным инструментом, который позволяет нам запускать Kubernetes локально. Minikube запускает одноузловой кластер Kubernetes внутри виртуальной машины на вашем компьютере.

```
curl -LO https://storage.googleapis.com/minikube/rele ases/latest/
minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Установка Minikube

```
minikube start
```

Приведенная выше команда запускает одноузловой кластер Kubernetes. Вы можете проверить его состояние с помощью команды `minikube status`.

Установка kubectl

kubectl — это инструмент для работы с Kubernetes из командной строки. Установите его, выполнив следующие действия:

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/a md64/kubectl"
chmod +x kubectl
sudo mv kubectl /usr/local/bin/
```

Развертывание университетского приложения на Kubernetes

А теперь развернем Docker-контейнер, созданный для университетского приложения, с помощью Kubernetes, выполнив описанные далее действия.

Создание конфигурации развертывания

Создайте файл **deployment.yaml**, описывающий требуемое состояние развертывания:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: university-app-deployment
spec:
replicas: 2
selector:
matchLabels:
app: university-app
template:
metadata:
labels:
app: university-app
spec:
containers:
- name: university-app
image: university-app:latest
ports:
- containerPort: 5000
```

Эта конфигурация предписывает Kubernetes поддерживать две копии университетского приложения, обеспечивая их полную доступность.

Развертывание приложения

Для применения конфигурации используйте **kubectl**:

```
kubectl apply -f deployment.yaml
```

Открытие приложения

Чтобы сделать университетское приложение доступным извне виртуальной сети Kubernetes, необходимо создать службу:

```
apiVersion: v1
kind: Service
metadata:
name: university-app-service
spec:
type: LoadBalancer
ports:
- port: 80
targetPort: 5000
selector:
app: university-app
```

Примените и эту конфигурацию:

```
kubectl apply -f service.yaml
```

Доступ к приложению

Поскольку будет использоваться Minikube, включите службу Minikube:

```
minikube service university-app-service
```

С помощью этой команды ваш браузер по умолчанию откроет службу, которая позволит вам взаимодействовать с развернутым приложением. Kubernetes упрощает и повышает эффективность развертывания, масштабирования и управления университетским приложением, позволяя ему выдерживать изменяющиеся нагрузки.

CI/CD для внутренних приложений на Python

Основные принципы работы CI/CD

Задачей непрерывной интеграции и развертывания (CI/CD или Continuous Integration and Continuous Deployment) является создание высококачественного программного обеспечения, которое можно легко использовать и адаптировать к потребностям пользователей, автоматизируя тестирование и развертывание изменений кода на регулярной основе. Чтобы обеспечить своевременное выявление ошибок интеграции, в ходе непрерывной интеграции выполняется автоматическое тестирование при каждом слиянии, а рабочие копии всех разработчиков сливаются в общий канал несколько раз в день. При непрерывном развертывании при условии успешного выполнения тестов приложение автоматически развертывается в производственной среде после каждого нового релиза.

Для приложений внутренней части, особенно созданных на языке Python и работающих в контейнерах Docker, как наше университетское приложение, CI/CD обеспечивает безопасный переход новых функций, исправлений и обновлений из среды разработки в производственную среду без сбоев в работе сервиса. Это очень важно для поддержания надежности и доступности внутренних сервисов, которые должны обрабатывать потенциально тысячи или даже миллионы запросов.

Использование CI/CD для университетского приложения

Учитывая контейнерную структуру нашего университетского приложения, для реализации надежного конвейера CI/CD мы можем использовать такие инструменты, как Jenkins, GitLab CI/CD или GitHub Actions.

Настройка контроля версий

Проверьте, размещена ли ваша кодовая база на платформе для управления версиями, например GitHub, GitLab или Bitbucket.

Эта платформа станет отправной точкой для вашего конвейера CI/CD.

Выбор инструмента CI/CD

В этом примере мы будем использовать GitHub Actions, который напрямую интегрируется с репозиториями GitHub.

Сформируйте рабочий процесс GitHub Actions:

1. Перейдите в свой репозиторий GitHub.
2. Перейдите на вкладку **Действия** (Actions) и создайте новый рабочий процесс.
3. Используйте стартовый шаблон для приложений Python или начните с нуля.

Настройте файл рабочего процесса. Создайте файл **.github/workflows/pythonapp.yml** и выполните следующие действия:

```
name: Python application CI/CD
on:
push:
branches: [ master ]
pull_request:
branches: [ master ]
jobs:
build:
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v2
- name: Set up Python 3.8
uses: actions/setup-python@v2
with:
python-version: 3.8
- name: Install dependencies
run: |
python -m pip install --upgrade pip
pip install flake8 pytest
if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
- name: Lint with flake8
run: |
```

```
# остановить сборку при наличии синтаксических ошибок Python
# или неопределенных имен
flake8 . --count --select=E9,F63,F7,F82 -- show-source --statistics
# exit-zero воспринимает все ошибки как предупреждения.
# В строке редактора GitHub используется 127 символов.
flake8 . --count --exit-zero --maxcomplexity= 10 --max-line-
length=127 -- statistics
- name: Test with pytest
run: |
pytest
deploy:
needs: build
runs-on: ubuntu-latest
if: github.ref == 'refs/heads/master' && github.event_name == 'push'
steps:
- uses: actions/checkout@v2
- name: Build Docker image
run: docker build . -t university-app:${{ github.sha }}
- name: Push Docker image to Registry
run: |
echo ${{ secrets.DOCKER_PASSWORD }} | docker login -u ${{
secrets.DOCKER_USERNAME }} --passwordstdin
docker push university-app:${{ github.sha }}
- name: Deploy to Kubernetes
run: kubectl set image deployment/university-app
universityapp=university-app:${{ github.sha }}
env:
KUBECONFIG: ${{ secrets.KUBECONFIG }}
```

В данном рабочем процессе определены два задания:

- *сборка* — здесь устанавливаются зависимости, с помощью flake8 выполняется линтинг и проводятся тесты с использованием pytest. В результате приложение собирается и проходит все тесты;
- *развертывание* — выполнение этого задания зависит от успешного выполнения задания по сборке. Здесь создается образ Docker, который размещается в реестре Docker, а затем выполняется обновление развертывания Kubernetes для использования нового образа.

Примечание переводчика

Flake8 — это утилита командной строки для контроля стиля кода в проектах на Python.

Линтинг (от *англ.* linting) — это автоматический поиск программных и стилистических ошибок в коде, оценка его качества.

Pytest — это фреймворк для тестирования кода на Python.

Для того чтобы поддерживать работоспособность, быстродействие и безопасность внутренних приложений, таких как наше университетское приложение, конвейер CI/CD просто необходим. Программное обеспечение может автоматически, быстро и безопасно доставляться благодаря интеграции методов CI/CD, особенно в контейнерной среде.

Использование Nginx в качестве обратного прокси-сервера

Функция Nginx

Nginx — это высокопроизводительный веб-сервер, который также широко используется в качестве обратного прокси и балансировщика нагрузки. Как обратный прокси-сервер, Nginx может управлять входящим трафиком и распределять его по различным внутренним серверам на основе правил конфигурации, улучшая масштабируемость, безопасность и производительность приложений.

Примечание переводчика

Обратный прокси-сервер (reverse proxy) — это тип прокси-сервера, который ретранслирует запросы клиентов из внешней сети на один или несколько серверов, логически расположенных во внутренней сети.

Балансировщик нагрузки (load balancer) — это сервис для распределения запросов между серверами кластера. Он позволяет сохранить доступность ресурса даже при аномальной нагрузке.

С точки зрения обратного прокси Nginx выступает в роли посредника для запросов клиентов, ищущих ресурсы на серверах. Он перехватывает все запросы к серверу до того, как они попа-

дают в приложение, что может обеспечить несколько преимуществ:

- Nginx может распределять входящий сетевой трафик между несколькими внутренними серверами, чтобы оптимизировать использование ресурсов, увеличить пропускную способность, сократить время отклика и обеспечить надежность;
- будучи обратным прокси, Nginx также может повысить безопасность, функционируя в качестве шлюза для серверов в Интернете. Он может выполнять такие задачи, как завершение протоколов SSL/TLS, шифруя и расшифровывая запросы клиентов и ответы серверов, тем самым снимая нагрузку с серверов приложений;
- Nginx также может кешировать содержимое сервера, сокращая количество запросов к серверу приложений, что улучшает время отклика для конечных пользователей.

Установка и настройка Nginx

Ниже описано, как установить и настроить Nginx.

Установка Nginx

На системах Ubuntu или Debian вы можете установить Nginx непосредственно из менеджера пакетов:

```
sudo apt update
sudo apt install nginx
```

После установки следует запустить службу Nginx и включить ее запуск при загрузке:

```
sudo systemctl start nginx
sudo systemctl enable nginx
```

Настройка Nginx в качестве обратного прокси-сервера

Чтобы настроить Nginx в качестве обратного прокси для университетского приложения, вам нужно изменить конфигурационные файлы Nginx, которые обычно находятся в каталоге **/etc/nginx/sites-available/**.

Далее приведена базовая конфигурация, которая настраивает Nginx в качестве обратного прокси.

1. Создайте новый файл конфигурации в каталоге **/etc/nginx/sites-available/** под названием **universityapp**.

```
sudo nano /etc/nginx/sitesavailable/university-app
```

2. Добавьте в файл следующую конфигурацию обратного прокси:

```
server {
listen 80;
server_name university.gitforgits.com;
location / {
# Предполагаем, что Flask-приложение работает с портом 5000
proxy_pass http://localhost:5000;
proxy_http_version 1.1;
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection 'upgrade';
proxy_set_header Host $host;
proxy_cache_bypass $http_upgrade;
}
}
```

Эта конфигурация предписывает Nginx прослушивать HTTP-запросы порта 80 для домена **university.gitforgits.com** и передавать их приложению, созданному на основе фреймворка Flask, запущенному на том же сервере, на порт 5000.

3. Активируйте конфигурацию — переместите файл конфигурации из каталога **sites-available** в каталог **sites-enabled**, чтобы разрешить его использование, а затем проверьте конфигурацию на наличие ошибок:

```
sudo ln -s /etc/nginx/sitesavailable/university-app
/etc/nginx/sitesenabled/
sudo nginx -t
```

Примечание переводчика

Каталог **sites-available** содержит файлы конфигураций виртуальных хостов. Каждый отдельный файл хранит информацию об определенном сайте. Это его имя, IP-адрес и другие данные. Каталог **sites-enabled**, в свою очередь, состоит только из конфигураций активных сайтов. Только из директории **sites-enabled** чита-

ются файлы конфигурации для виртуальных хостов. Также в ней хранятся ссылки на **sites-available**.

4. Если нет сообщений об ошибках, перезапустите Nginx, чтобы применить изменения:

```
sudo systemctl restart nginx
```

После всего этого Nginx будет обрабатывать запросы к **university.gitforgits.com** и перенаправлять их приложению на основе Flask, запущенному локально на порту 5000. Более того, использование сервера Nginx в качестве обратного прокси добавляет уровень абстракции к сетевому взаимодействию с вашими внутренними серверами. Это может обеспечить дополнительные меры безопасности, такие как белые списки IP-адресов, ограничение скорости и обработка политик CORS (Cross-Origin Resource Sharing или совместное использование ресурсов).

SSL-сертификаты и настройка HTTPS

Описание SSL/TLS и HTTPS

Обеспечение безопасности в Интернете и аутентификация веб-сайтов в значительной степени зависят от сертификатов SSL (Secure Sockets Layer или уровень защиты сокетов). Данные, передаваемые между пользователями и веб-сайтом, шифруются и защищаются при настройке протокола HTTPS (Hypertext Transfer Protocol Secure или безопасный протокол передачи гипертекста) с использованием сертификатов SSL. Это особенно важно для веб-приложений, которые работают с конфиденциальной информацией, например для университетских приложений. Большинство браузеров отображают значок замка, чтобы показать, что соединение безопасно, а веб-сайты, использующие SSL, отображают https в своих URL-адресах.

Генерация SSL-сертификатов

Для работы с университетскими приложениями вам понадобится надежный SSL-сертификат, выданный центром сертификации (Certificate Authority, CA). Сейчас мы рассмотрим процесс полу-

чения сертификата с помощью Let's Encrypt, популярного бесплатного центра сертификации, и Certbot, инструмента, который упрощает получение и установку сертификатов.

Установка Certbot

Certbot — это простой в использовании автоматический клиент для поиска и установки SSL/TLS-сертификатов для вашего веб-сервера. Чтобы установить Certbot и его плагин для сервера Nginx на Linux-сервере, выполните следующие действия:

```
sudo apt-get update
sudo apt-get install software-propertiescommon
sudo add-apt-repository ppa:certbot/certbot
sudo apt-get update
sudo apt-get install certbot python3-certbot-nginx
```

Получение сертификата

Чтобы автоматически получать и устанавливать сертификаты, а также настраивать Nginx на их использование, запустите Certbot с плагином Nginx:

```
sudo certbot --nginx -d university.gitforgits.com
```

Замените адрес `university.gitforgits.com` на доменное имя вашего университетского приложения. Certbot изменит конфигурацию Nginx для вашего домена, чтобы использовать полученный SSL-сертификат, и настроит периодическое продление.

Проверка установки SSL-сертификата

Проверить правильность установки SSL-сертификатов можно, зайдя на свой домен через веб-браузер по адресу **https://**. Браузер должен показать, что соединение безопасно. Также для проверки конфигурации SSL на вашем сайте можно использовать такие онлайн-инструменты, как SSL Labs' SSL Test.

Конфигурация HTTPS в Nginx

Хотя Certbot автоматически настраивает Nginx на использование SSL-сертификата, представление о том, как вносятся изменения,

может помочь вам управлять конфигурацией. Ниже приведен типичный фрагмент конфигурации SSL в Nginx:

```
server {
listen 443 ssl http2;
server_name university.gitforgits.com;
ssl_certificate
/etc/letsencrypt/live/university.gitforgits.com/fullchain.pem;
ssl_certificate_key
/etc/letsencrypt/live/university.gitforgits.com/privkey.pem;
ssl_protocols TLSv1.2 TLSv1.3;
ssl_ciphers 'ECDHE-ECDSA-AES128-GCMSHA256: ECDHE-RSA-AES128-GCM-SHA256';
ssl_prefer_server_ciphers on;
ssl_session_cache shared:SSL:10m;
# Добавьте HSTS (HTTP Strict Transport Security)
add_header Strict-Transport-Security "maxage=31536000" always;
location / {
proxy_pass http://localhost:5000;
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;
}
}
```

Certbot не только упрощает настройку SSL/TLS-сертификатов для университетских приложений, но и делает их более надежными и заслуживающими доверия в глазах пользователей. Благодаря тому, что он работает с Nginx, приложение может использовать надежные средства защиты, которые предоставляются одним из самых популярных веб-серверов.

Масштабирование приложений на Python

Необходимость масштабирования

Чтобы успешно выполнить масштабирование приложений, написанных на языке Python, необходимо изменить код таким образом, чтобы он мог работать с большим количеством пользователей без ущерба для производительности. Например, в пиковые

моменты регистрации тысячи пользователей могут пытаться получить доступ к системе одновременно, поэтому университетскому приложению придется выполнять несколько сценариев нагрузки. Благодаря способности масштабироваться приложение может динамически управлять своими ресурсами, чтобы эффективно справляться с такими пиковыми нагрузками.

Горизонтальное и вертикальное масштабирование

Горизонтальное масштабирование (масштабирование наружу/внутрь)

Горизонтальное масштабирование — это добавление дополнительных компьютеров или экземпляров приложения к вашему пулу ресурсов, чтобы справиться с возросшей нагрузкой. Это особенно эффективно для распределенных систем и хорошо поддерживается системами управления контейнерами, такими как Kubernetes.

Вертикальное масштабирование (масштабирование вверх/вниз)

Этот метод предполагает добавление дополнительной вычислительной мощности (процессора, оперативной памяти) к существующим машинам, но ограничен физическими возможностями оборудования. Для большинства веб-приложений горизонтальное масштабирование предпочтительнее из-за его гибкости и согласованности с облачными ресурсами.

Реализация масштабирования в Kubernetes

Поскольку приложение Kubernetes уже выполняет задачу контейнерной обработки, мы можем воспользоваться возможностями Kubernetes для горизонтального масштабирования нашего приложения. С помощью функции, известной как Horizontal Pod Autoscaler (HPA), Kubernetes может динамически изменять количество запущенных капсул (подсистем) в зависимости от нагрузки на систему.

Определение запросов и лимитов ресурсов

Прежде чем настраивать функцию автоматического масштабирования (HPA), определите запросы и ограничения ресурсов в конфигурации развертывания. Это очень важно для Kubernetes, чтобы принимать обоснованные решения о том, когда следует увеличить или уменьшить масштаб. Затем обновите **deployment.yaml** с запросами и лимитами ресурсов, как показано ниже:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: university-app-deployment
spec:
replicas: 2
selector:
matchLabels:
app: university-app
template:
metadata:
labels:
app: university-app
spec:
containers:
- name: university-app
image: university-app:latest
resources:
requests:
cpu: "500m"
memory: "256Mi"
limits:
cpu: "1000m"
memory: "512Mi"
ports:
- containerPort: 5000
```

В соответствии с этими настройками для работы каждого экземпляра приложения требуется не менее 500 m CPU и 256 Mi памяти, а максимальный предел составляет 1000 m CPU и 512 Mi памяти.

Примечание переводчика

500 m CPU в контексте Kubernetes означает запрос на 500 милли-CPU или 500 миллисекунд процессорной мощности. Это значение указывает на то, что контейнер запрашивает определенное количество вычислительных ресурсов в абсолютных величинах, независимо от того, на каком процессоре он запущен — одноядерном, двухъядерном или 48-ядерном процессоре.

256 Mi памяти означает 256 мебибайт оперативной памяти.

Мегабайты обозначаются как Mb, так и Mi из-за разных определений этой единицы измерения. Mb (мегабит) используется для обозначения объема данных в битах. Mi (мебибайт) применяется для обозначения емкости дисков. Раньше производители считали, что в 1 мегабайте — 1000 килобайт. Но оказалось, что их не 1000 а 1024. Таким образом, Mb обозначает мегабиты, а Mi — мебибайты.

Мебибайт (русское обозначение — МиБ; международное — MiB) — это единица измерения количества информации, равная 10 242 байтам. Мебибайт был разработан, чтобы заменить термин «мегабайт» в тех областях информатики, в которых он означал 1 048 576 байт. Он больше мегабайта на 48 576 байт (более чем на 48 килобайт или 47 кибибайт).

https://ru.wikipedia.org/wiki/Мебибайт
https://ru.ruwiki.ru/wiki/Мебибайт

Настройка горизонтального автоматического масштабирования подсистем

Создайте ресурс HPA, ориентированный на ваше развертывание. HPA регулирует количество подов (запущенных экземпляров приложения) при развертывании в зависимости от загрузки процессора.

```
kubectl autoscale deployment university-appdeployment
--cpu-percent=50 --min=2 -- max=10
```

С помощью этой команды Kubernetes устанавливает целевой уровень загрузки процессора в 50% для каждого пода. Если средняя загрузка процессора превышает этот порог, Kubernetes запускает новые поды (но не более 10: `max=10`), чтобы справиться с нагрузкой.

Примечание переводчика

HPA (Horizontal Pod Autoscaler) — это метод автоскейлинга в Kubernetes, который автоматически регулирует количество подов (число

экземпляров приложения, запущенных в кластере) в приложении в зависимости от текущей нагрузки. HPA основывается на метриках использования ресурсов, таких как CPU и память. Он следит за этими метриками и, если они выходят за установленные пределы, увеличивает или уменьшает количество подов в приложении. HPA функционирует как контроллер, который периодически проверяет текущие метрики и сравнивает их с целевыми значениями, заданными в конфигурации. Если метрики отклоняются от целевых значений, HPA вносит изменения в количество подов.

Автоскейлинг в Kubernetes — это процесс автоматического увеличения или уменьшения количества ресурсов, выделенных для приложения, в зависимости от текущей нагрузки.

Контроль масштабирования

Вы можете отслеживать действия по масштабированию, выполняемые Kubernetes, с помощью команды:

```
kubectl get hpa
```

Такой подход позволяет эффективно использовать гибкость облачных ресурсов и гарантирует, что приложение останется работоспособным независимо от количества запросов пользователей.

Резюме

В этой главе мы рассмотрели, как масштабировать внутренние приложения, созданные на основе языка Python, уделив особое внимание использованию современных инструментов и методологий для безопасного и эффективного управления приложениями. В начале главы мы познакомились с программной платформой Docker и технологией контейнеров, сделав акцент на том, как Docker упрощает развертывание, заключая приложения в контейнеры. Это позволяет легко контролировать и откатывать версии, а также обеспечивает согласованность в различных средах.

Затем мы занялись изучением Kubernetes, платформы для организации эффективной работы с контейнерами. Стало ясно, что эта платформа критически важна для горизонтального масштабирования приложений, что означает, что приложения могут управлять возросшей нагрузкой, распределяя экземпляры приложений по кластеру серверов. Для университетского приложения в пе-

риоды регистрации студентов и другие моменты изменения нагрузки этот метод был особенно полезен для динамического масштабирования университетского приложения.

Далее в главе рассматривается управление контейнерами, а также передовые методы технологии CI/CD (Continuous Integration and Continuous Deployment или непрерывная интеграция и непрерывное развертывание). Повышение эффективности и надежности производственных обновлений — результат автоматизации тестирования и развертывания приложений с помощью этих методов.

Благодаря внедрению конвейеров, непрерывной интеграции и непрерывного развертывания с помощью таких инструментов, как GitHub Actions, любые изменения, вносимые в приложение, автоматически встраиваются, тестируются и развертываются, исключая возможность человеческой ошибки. Кроме того, речь шла об использовании сервера Nginx в качестве обратного прокси. С помощью Nginx клиентские запросы эффективно распределяются, безопасность повышается, а производительность приложения увеличивается за счет кеширования и балансировки нагрузки.

В конце главы было рассказано о конкретных приемах масштабирования приложений, а также о SSL-сертификатах и настройке протокола HTTPS для защиты соединений. Подводя итог, можно сказать, что в этой главе было продемонстрировано, как масштабировать университетское приложение с помощью Kubernetes, приведены практические шаги по масштабированию и показано, как добиться автоматического масштабирования для поддержания оптимальной производительности независимо от колебаний нагрузки.

ГЛАВА 9
Микросервисы и интеграция с облаком

Введение

В этой главе рассматриваются основные принципы развертывания, эксплуатации и проектные решения, используемые в современных внутренних приложениях, написанных на языке Python. Также мы рассмотрим такие перспективные направления, как облачная интеграция и микросервисы. Для начала в главе объясняется, как использовать Python для проектирования и создания микросервисов, и демонстрируется, как разделить большое приложение на небольшие, независимо развертываемые сервисы. Этот метод не только позволяет улучшить масштабируемость приложения и привносит дополнительные удобства в обслуживание, но и хорошо подходит для постоянно меняющегося облачного ландшафта.

Далее мы расскажем, как для управления этими микросервисами использовать программное обеспечение Docker и Kubernetes. Эти инструменты предлагают эффективные решения для управления приложениями и их упаковки в контейнеры, что упрощает организацию, масштабирование и поддержку микросервисов. В этой главе мы рассмотрим, как Kubernetes автоматизирует развертывание, масштабирование и работу контейнерных служб, а контейнеры Docker обеспечивают управление зависимостями и средами микросервисов.

Далее мы рассмотрим самые передовые способы использования коммерческого публичного облака Amazon Web Services (AWS) для размещения приложений, написанных на языке Python. Вы узнаете о сервисах и ресурсах, доступных в облаке, и о том, как с их помощью можно повысить производительность и эффектив-

ность ваших приложений. В главе также представлена идея бессерверных архитектур с акцентом на платформенной услуге AWS Lambda и ее использовании для запуска кода независимо от серверов. Для микросервисов, которые запускаются нечасто или асинхронно, эта модель может стать экономически эффективным вариантом, и она отлично подходит для приложений с непредсказуемой рабочей нагрузкой.

Наконец, в главе рассказывается о том, как реализовать систему удаленного вызова процедур gRPC для межсервисного взаимодействия. Эффективное взаимодействие между микросервисами является главной задачей этого высокопроизводительного фреймворка с открытым исходным кодом, который благодаря поддержке множества языков программирования наилучшим образом подходит для полиглот-среды. В этой главе читатели узнают, как эффективно проектировать, создавать и управлять архитектурами микросервисов, используя самые современные контейнерные технологии, стратегии развертывания облаков и протоколы взаимодействия. Конечный результат — масштабируемые, надежные и эффективные внутренние системы.

Проектирование и разработка микросервисов с помощью Python

С помощью процесса проектирования и создания микросервисов можно одно целое приложение разделить на более мелкие, независимо развертываемые сервисы. Сервисы работают независимо друг от друга и обмениваются данными через простые протоколы, в большинстве случаев HTTP ресурсного API. Приложения, подобные университетским системам, значительно выигрывают от такого подхода, поскольку различные модули (например, модуль, отвечающий за работу приемной комиссии, управления курсами и за учет студентов) могут функционировать практически независимо друг от друга.

Разбиение приложения на микросервисы

Чтобы преобразовать университетское приложение в архитектуру микросервисов, необходимо выделить логические модули, которые могут функционировать как независимые сервисы.

Рассмотрим базовую разбивку:

1. Служба приема — занимается выполнением всех функций, связанных с обработкой поданных заявлений и приемом студентов.
2. Служба управления курсами — осуществляет управление учебными программами, регистрацией и назначением преподавателей.
3. Служба учета студентов — ведет учет студентов, включая оценки и личную информацию.
4. Служба аутентификации — управляет аутентификацией и авторизацией пользователей во всех службах.

Определение границ сервисов

Каждый микросервис должен содержать собственную доменную логику и данные, чтобы обеспечить свободное взаимодействие и высокую степень связности. Например:

- служба приема должна управлять базами данных, связанными с формами заявлений и оценками абитуриентов;
- служба управления курсами контролирует каталог курсов и данные о зачислении;
- служба учета студентов управляет базами данных, содержащими оценки и биографические данные студентов.

Создание независимых сред

Каждая служба должна иметь возможность работать независимо как во время разработки, так и во время развертывания. Для этого необходимо создать отдельные среды разработки, репозитории контроля версий и баз данных для каждого сервиса.

Разработка API для межсервисного взаимодействия

Сервисы должны взаимодействовать друг с другом с помощью интерфейсов обмена информацией (API). Для создания микросервисов часто выбирают архитектурный стиль REST из-за его простоты и хорошей интеграции с Интернетом. Ниже приведен пример конечной точки API для Службы управления курсами:

```
from flask import Flask, jsonify, request
app = Flask(__name__)
# Имитация обращения к базе данных
courses = {
'CS101': {'title': 'Introduction to Computer Science', 'capacity':
30, 'enrollment': 0}
}
@app.route('/courses/<course_id>', methods=['GET'])
def get_course(course_id):
course = courses.get(course_id)
if course:
return jsonify(course), 200
else:
return jsonify({"message": "Course not found"}), 404
@app.route('/courses/<course_id>/enroll', methods=['POST'])
def enroll_student(course_id):
course = courses.get(course_id)
if course and course['enrollment'] < course['capacity']:
course['enrollment'] += 1
return jsonify(course), 200
else:
return jsonify({"message": "Enrollment failed"}), 400
if __name__ == '__main__':
app.run(port=5002)
```

Упаковка сервисов в контейнеры

Каждый сервис, чтобы обеспечить его стабильную работу в различных средах, должен быть упакован с помощью Docker в отдельный контейнер. Ниже приведен пример файла Dockerfile, предназначенного для службы управления курсами:

```
FROM python:3.8-slim
WORKDIR /app
COPY . /app
RUN pip install flask
EXPOSE 5002
CMD ["python", "app.py"]
docker build -t course-management-service .
docker run -p 5002:5002 coursemanagement-service
```

Службы работают по отдельности, но для работы приложения в целом их необходимо объединить. Для объединения иногда используется API-шлюз, который направляет через себя запросы к нужным сервисам. Благодаря такому модульному подходу улучшается масштабируемость приложения и гораздо легче поддерживать его работу. Также повышается гибкость процессов разработки и развертывания.

Управление микросервисами с помощью Docker и Kubernetes

Для эффективного управления микросервисами необходима надежная стратегия контейнерной обработки и управления созданными контейнерами. Это означает, что каждый микросервис должен быть независимым, масштабируемым, с обеспечением хорошей поддержки его работы. С помощью Docker и Kubernetes такое управление становится возможным, поскольку позволяет развертывать сервисы в виде контейнеров, которые могут динамически масштабироваться и управляться с помощью Kubernetes.

Контейнеризация с помощью Docker

Docker позволяет упаковать каждый компонент архитектуры микросервисов вашего университетского приложения в отдельный контейнер. Таким образом обеспечивается не только защита упаковываемых в контейнер компонентов, но и среды выполнения, минимизируются конфликты между сервисами и упрощаются процессы развертывания.

Упаковка каждого микросервиса в контейнер

Для каждого микросервиса в университетской системе («Приемная комиссия», «Управление курсами», «Учет студентов», «Аутентификация») необходимо создать отдельный файл Dockerfile, определяющий способ создания сервиса.

Например, Dockerfile для сервиса «Управление курсами» может выглядеть следующим образом:

```
# Используйте официальную среду выполнения Python в качестве
# основного образа
FROM python:3.8-slim
# Установите рабочий каталог в контейнере
WORKDIR /app
# Скопируйте содержимое текущего каталога в контейнер по адресу /app
COPY . /app
# Установите все необходимые пакеты, указанные
# в файле requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt
# Сделайте порт 5002 доступным для мира за пределами этого контейнера
EXPOSE 5002
# Определите переменную окружения
ENV NAME World
# При запуске контейнера необходимо запустить файл app.py
CMD ["python", "app.py"]
```

Создайте и запустите контейнер Docker с помощью команд, аналогичных приведенным ниже:

```
docker build -t course-management-service .
docker run -p 5002:5002 coursemanagement-service
```

Организация работы с помощью Kubernetes

Kubernetes отлично справляется с управлением подобными контейнерными приложениями, особенно в архитектуре микросервисов, где высокая доступность, масштабируемость и обнаружение сервисов имеют решающее значение.

Создание развертываний Kubernetes

Для каждого микросервиса должно использоваться свое развертывание Kubernetes, которое описывает, сколько реплик (экземпляров) этого сервиса должно существовать, как они настраиваются и как происходит распространение обновлений.

Ниже приведен пример развертывания Kubernetes для сервиса «Управление курсами»:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: course-management-deployment
spec:
replicas: 3
selector:
matchLabels:
app: course-management
template:
metadata:
labels:
app: course-management
spec:
containers:
- name: course-management
image: course-management-service:latest
ports:
- containerPort: 5002
```

При этом развертывании Kubernetes настраивается таким образом, чтобы три экземпляра службы «Управление курсами» всегда были запущены.

Управление сервисами с помощью Kubernetes Services

Чтобы развернуть службу «Управление курсами» в кластере Kubernetes, определите службу Kubernetes:

```
apiVersion: v1
kind: Service
metadata:
name: course-management-service
```

```
spec:
type: ClusterIP
ports:
- port: 5002
targetPort: 5002
selector:
app: course-management
```

Этим гарантируется, что служба «Управление курсами» может быть обнаружена и доступна другим службам в кластере с помощью встроенного в Kubernetes сервера DNS. Такая настройка не только повышает эффективность работы, но и улучшает надежность и масштабируемость всей архитектуры университетских приложений.

Развертывание приложений Python на AWS

Используя коммерческое публичное облако Amazon Web Services (AWS) для развертывания приложений, написанных на языке Python, вы получаете доступ к масштабируемой облачной вычислительной среде, способной с легкостью справиться с различными требованиями, предъявляемыми к такому типу приложений, как университетские приложения. Облако AWS предоставляет широкий спектр услуг для развертывания и управления приложениями — от базовых экземпляров среды Elastic Compute Cloud (EC2) до более сложных управляемых сред с ECS и EKS.

Настройка AWS для университетского приложения

Перед развертыванием приложения необходимо создать учетную запись в облаке AWS и настроить необходимые службы.

Создание учетной записи AWS

Если вы еще этого не сделали, зарегистрируйте свою учетную запись AWS на **aws.amazon.com**. После создания учетной записи войдите в консоль управления AWS.

Настройка IAM (управление идентификацией и доступом)

Чтобы обеспечить максимальную безопасность, следует управлять ресурсами AWS с помощью системы IAM (Identity and Access Management или управление идентификацией и доступом):

1. Создайте нового пользователя IAM:
   - Перейдите на панель **IAM** в AWS Console.
   - Выберите раздел **Users** (Пользователи) на боковой панели и нажмите кнопку **Add user** (Добавить пользователя).
   - Введите имя пользователя и как тип доступа выберите **Programmatic access** (Программный доступ).
   - Следуя подсказкам, присоедините политики напрямую или добавьте пользователя в группу с соответствующими разрешениями.
2. Защитите корневую систему и пользователей IAM с помощью MFA (многофакторной аутентификации):
   - На панели **IAM** выберите ваше имя пользователя IAM.
   - На вкладке **Security credentials** (Учетные данные безопасности) рядом с пунктом **Assigned MFA device** (Назначенное устройство MFA) нажмите кнопку **Manage** (Управление) и, чтобы добавить MFA, далее следуйте инструкциям.

Настройка AWS CLI

Установите интерфейс AWS CLI (AWS Command Line Interface или интерфейс командной строки AWS) на свой локальный компьютер, чтобы взаимодействовать с сервисами AWS прямо с терминала:

```
pip install awscli
aws configure
```

При появлении запроса введите идентификатор ключа доступа AWS, секретный ключ доступа, регион и формат вывода.

Развертывание приложения на AWS

Чтобы автоматизировать процесс развертывания приложения, написанного на языке Python, вы можете использовать либо базовый экземпляр среды Elastic Beanstalk, либо среду EC2, которая позволяет получить дополнительный контроль.

Вариант 1: использование Elastic Beanstalk

1. AWS Elastic Beanstalk упрощает развертывание и масштабирование приложений.
   - Убедитесь, что в корневом каталоге вашего приложения имеется файл **requirements.txt**, в котором перечислены все необходимые пакеты Python.
   - Включите **Procfile**, чтобы указать команды, выполняемые контейнерами приложения при запуске, например **web: python app.py**.
2. Развертывание с помощью Elastic Beanstalk.
   - Установите Elastic Beanstalk CLI.
   - Перейдите в каталог вашего проекта и выполните команду:

```
eb init -p python-3.8 my-university-app -- region your-aws-region
eb create my-university-app-env
```

В результате будет настроена среда Elastic Beanstalk, которая управляет развертыванием, начиная с выделения мощностей, балансировки нагрузки, автоматического масштабирования и заканчивая мониторингом состояния приложений.

Вариант 2: использование EC2

Если вам требуется дополнительный контроль над сервером:

1. Запустите экземпляр среды EC2.
   - В консоли AWS перейдите на панель **EC2** и нажмите кнопку **Launch Instance** (Запустить экземпляр).
   - Выберите подходящий образ AMI (Amazon Machine Image или образ машины Amazon), например Ubuntu Server.

- Укажите тип экземпляра, настройте сведения об экземпляре, добавьте хранилище, добавьте метки, настройте группу безопасности.

2. Подключитесь по SSH к вашему экземпляру.
   - Когда экземпляр будет запущен, подключитесь к нему с помощью SSH с приватным ключом.

     ```
     ssh -i "your-key-pair.pem" ubuntu@ec2-yourinstance-public-dns.amazonaws.com
     ```

3. Настройте окружающую среду.
   - Установите Python, pip и другие зависимости.
   - Клонируйте репозиторий или перенесите файлы приложения в экземпляр.
   - Настройте веб-сервер, например Nginx или Apache, для обслуживания вашего приложения.

Независимо от того, предпочитаете ли вы контроль и гибкость, присущие среде EC2, или автоматизированные функции управления среды Elastic Beanstalk, облако AWS предоставляет надежные инструменты, которые помогут вам развернуть сложные приложения.

Использование бессерверных архитектур с AWS Lambda

Описание AWS Lambda

Мы можем максимально повысить эффективность и упростить развертывание некоторых функций в нашем университетском приложении, используя бессерверные архитектуры, в частности функцию «платформа как услуга» AWS Lambda, которая позволяет снизить, по сравнению с традиционными серверными установками, операционные накладные расходы и затраты на масштабирование. AWS Lambda — это вычислительный сервис, позволяющий запускать код без подготовки серверов или управления ими. Вы платите только за потребляемое вычислительное время — когда ваш код не выполняется, плата не взимается.

С помощью Lambda вы можете запускать код практически для любого типа приложений или внутренних служб с отсутствием администрирования. AWS Lambda выполняет ваш код только тогда, когда это необходимо, и автоматически масштабируется. Не требуя добавления или управления серверами, AWS Lambda позволяет запускать код в ответ на такие события, как HTTP-запросы через API Gateway, изменения данных в DynamoDB или изменения состояния в контейнерах S3.

Установка AWS Lambda

Чтобы воспользоваться сервисом AWS Lambda для части университетского приложения, сначала нужно настроить соответствующую среду AWS и подготовить приложение к развертыванию.

Подготовка приложения

Предположим, у нас есть функция Python для выполнения регистрации студентов, которая является хорошим кандидатом для бессерверной обработки благодаря своей событийной природе (например, запускается при отправке регистрационной формы).

Создайте файл **lambda_function.py**. Этот файл будет содержать функцию, выполняемую сервисом AWS Lambda.

```
import json
def lambda_handler(event, context):
# Предполагаем, что в "событии" содержатся регистрационные данные
student = event['body']
# Обработка регистрации
registration_status = process_registration(student)
return {
'statusCode': 200,
'body': json.dumps({
'message': Регистрация прошла успешно',
'details': registration_status
})
}
def process_registration(student):
# Логика регистрации студента
return "Студент зарегистрирован."
```

Если ваша функция зависит от внешних библиотек, упакуйте их вместе с пакетом развертывания:

```
pip install --target ./package requests
cd package
zip -r ../my-deployment-package.zip .
cd ..
zip -g my-deployment-package.zip lambda_function.py
```

Создание функции Lambda в AWS

Войдите в консоль управления AWS Management Console и откройте консоль AWS Lambda.

1. Создайте новую лямбда-функцию:
   - Нажмите **Create function** (Создать функцию).
   - Выберите **Author from scratch** (Создать с нуля).
   - Введите имя функции.
   - Выберите Python 3.8 в качестве среды выполнения.
   - Настройте права доступа, выбрав существующую роль или создав новую, которая будет иметь права доступа к ресурсам, необходимым вашей функции.
2. Загрузите свой код:
   - В разделе **Function code** (Код функции) загрузите свой zip-файл (`my-deploymentpackage.zip`).
   - Введите информацию об обработчике (например, `lambda_function.lambda_handler`).
3. Настройте триггер:
   - Нажмите кнопку **Add trigger** (Добавить триггер).
   - Выберите **AWS API Gateway**.
   - Создайте новый API или подключитесь к существующему.
   - Для тестирования установите уровень безопасности **Open** (Открытый).

4. Развертывание и тестирование:
   - Разверните изменения.
   - Используйте предоставленный URL-адрес шлюза API для запуска вашей функции с помощью HTTP-запросов.

AWS Lambda может обрабатывать различные события в университетском приложении, такие как отправка уведомлений студентам, обработка запросов на регистрацию или интеграция с другими сервисами AWS, такими как DynamoDB для хранения данных или S3 для управления файлами.

Каждая из этих функций может быть внедрена в отдельные функции Lambda, что делает систему модульной и снижает нагрузку на основной сервер приложений. Это не только улучшает масштабируемость, позволяя каждой функции масштабироваться независимо от спроса, но и оптимизирует расходы, поскольку вы платите только за используемое вычислительное время.

Реализация gRPC для взаимодействия микросервисов

Зачем нужен gRPC

В архитектуре микросервисов для общей производительности и надежности приложения очень важна эффективная связь между сервисами. gRPC, разработанный Google, — это высокопроизводительный универсальный фреймворк RPC (Remote Procedure Call или удаленный вызов процедур) с открытым исходным кодом, который использует протокол HTTP/2 для транспорта и Protocol Buffers в качестве интерфейса. Этот протокол особенно хорошо подходит для подключения сервисов в виде микросервисов благодаря поддержке нескольких языков и эффективности подключения полиглот-сервисов в распределенных системах.

> ***Примечание переводчика***
>
> Protocol Buffers (Буферы протокола) — это протокол сериализации данных, разработанный компанией Google. Он используется для эффективного и быстрого обмена структурированными данными

между разными компьютерными системами, языками программирования и платформами.

Использование традиционного протокола программного интерфейса RESTful API и передачи данных через протокол HTTP/1.1 может стать узким местом из-за их многословного текстового формата и накладных расходов на открытие и закрытие соединений. gRPC решает эти проблемы за счет использования протокола HTTP/2, который поддерживает многократное выполнение большого количества запросов через одно соединение и передачи данных в двоичном формате, что значительно ускоряет сериализацию и десериализацию.

Реализация gRPC в микросервисах на Python

Чтобы внедрить фреймворк gRPC в университетское приложение, где мы уже создали несколько микросервисов, таких как прием, управление курсами и записи студентов, выполните описанные далее действия.

Определение сервиса с помощью буферов протокола

Сначала с помощью буферов протокола определите службу gRPC, а для методов — типы запросов и ответов. Создайте файл **.proto** для каждого взаимодействия со службой.

Например, создадим простую службу для микросервиса «Управление курсами»:

```
// course_management.proto
syntax = "proto3";
package coursemanagement;
// Определение микросервиса "Управление курсами"
service CourseManagement {
// Посылает приветствие
rpc GetCourseDetails (CourseRequest) returns (CourseResponse) {}
}
// Запрос с указанием идентификатора курса
message CourseRequest {
string course_id = 1;
}
```

```
// Ответное сообщение с деталями курса
message CourseResponse {
string course_id = 1;
string title = 2;
string description = 3;
int32 credits = 4;
}
```

Создание кода сервера и клиента

Используйте компилятор буфера протокола для создания заглушек клиента и сервера из ваших файлов **.proto**.

```
python -m grpc_tools.protoc -I. -- python_out=. --grpc_python_out=.
course_management.proto
```

Эта команда генерирует файлы **course_management_pb2.py** и **course_management_pb2_grpc.py**, в которых содержатся классы для отправки запросов и ответов, а также классы сервера и клиента соответственно.

Реализация сервиса на Python

Реализуйте логику обработки запросов gRPC на стороне сервера.

```
from concurrent import futures
import grpc
import course_management_pb2
import course_management_pb2_grpc
class CourseManagementServicer(course_manage
                               ment_pb2_grpc.CourseManagementServicer):
def GetCourseDetails(self, request, context):
response = course_management_pb2.CourseResponse()
course_id = request.course_id
# Имитация получения сведений о курсе из базы данных.
response.course_id = course_id
response.title = "Introduction to Microservices"
response.description = "Learn the basics of microservices
                                                    architecture."
response.credits = 3
return response
def serve():
server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
```

```
course_management_pb2_grpc.add_CourseManagementServicer_to_server(
CourseManagementServicer(), server)
server.add_insecure_port('[::]:50051')
server.start()
server.wait_for_termination()
if __name__ == '__main__':
serve()
```

Создание клиента

Код клиента будет вызывать службу, определенную сервером.

```
import grpc
import course_management_pb2
import course_management_pb2_grpc
def run():
with grpc.insecure_channel('localhost:50051') as channel:
stub = course_management_pb2_grpc.CourseManagementStub(channel)
response = stub.GetCourseDetails(course_management_pb2
                                 .CourseRequest(course_id='CS101'))
print("Курс получен: ", response.title)
if __name__ == '__main__':
run()
```

Протокол gRPC необходим для обеспечения согласованности данных и повышения эффективности работы в архитектуре микросервисов. В приведенном выше примере показано, как настроить простой gRPC-сервер и клиент внутри микросервисов, которые можно использовать в качестве отправной точки при добавлении в приложение дополнительных сервисов.

Резюме

В этой главе мы рассмотрели перспективные идеи создания микросервисов и облачной интеграции, уделив особое внимание оптимизации архитектуры Python и стратегии развертывания университетского приложения. Первым шагом было планирование и создание микросервисов. Компоненты университетского приложения, предназначенные для приема, управления курсами и

учета студентов, были разделены на более мелкие независимые сервисы. Такой подход облегчил управление и масштабирование.

Управление этими микросервисами осуществлялось с помощью Docker и Kubernetes, демонстрируя, как использование контейнеров упрощает развертывание и организацию микросервисов. Для обеспечения мобильности и согласованности в разных средах мы с помощью Docker поместили каждый микросервис в отдельный контейнер. Управление этими контейнерами, включая управление их жизненным циклом, масштабирование и сохранение состояния, осуществлялось с помощью Kubernetes.

Облачные среды, и особенно облачный сервис AWS, заняли центральное место в стратегиях развертывания. В главе были продемонстрированы такие полезные сервисы облака AWS, как Elastic Beanstalk и Elastic Compute Cloud (EC2), которые позволяют автоматизировать развертывание Python-приложений и предоставляют больше возможностей для контроля и настройки. Кроме того, мы узнали, как использовать вычислительный сервис AWS Lambda в бессерверных архитектурах. Этот метод повышает эффективность работы, позволяя выполнять определенные функции приложения, такие как обработка событий регистрации, без необходимости выделения и масштабирования серверов.

И в заключение в главе рассматривается, как можно использовать фреймворк gRPC для удаленного вызова процедур, что способствует эффективному взаимодействию между микросервисами. Являясь эффективной заменой традиционных программных интерфейсов REST API, фреймворк gRPC улучшил коммуникацию между распределенными сервисами за счет использования протокола HTTP/2 и буферов протоколов. В целом в этой главе мы узнали, как с помощью современных инструментов и методик спроектировать и построить надежную, масштабируемую внутреннюю часть веб-приложений, чтобы университетское приложение могло адаптироваться к меняющимся нагрузкам и оставаться работоспособным в любых условиях.

ГЛАВА 10
Брокеры сообщений и асинхронная обработка задач

Введение

В этой главе мы рассмотрим, по какой причине брокеры сообщений и асинхронная обработка задач являются важнейшими компонентами современных архитектур приложений. Особенно для работы с масштабируемыми приложениями и операциями, требующими высокой надежности их выполнения. В этой главе подробно рассматривается широкий спектр методов и ресурсов, необходимых для эффективного взаимодействия приложений, работающих на асинхронной основе, и для управления проектами.

В начале главы мы в общих чертах расскажем о брокерах сообщений, как они обеспечивают связь и передачу данных между компонентами приложений и почему брокеры сообщений так важны для разделения этих компонентов. Такая конфигурация повышает адаптивность, масштабируемость и позволяет разделить приложение на модули.

Далее, после вводной части мы познакомимся с Kafka — известной платформой для распределенной потоковой передачи данных. Эта платформа особенно востребована при обработке данных в режиме реального времени. Здесь объясняется, как университетское приложение может использовать Kafka для управления высокопроизводительными потоками данных, что позволит проводить аналитику и принимать решения в режиме реального времени. После мы рассмотрим Celery, распределенную систему очередей задач, которая при обработке фоновых задач взаимодействует с брокером сообщений RabbitMQ и сетевым журналируемым хранилищем данных и брокером сообщений Redis, по-

скольку далее речь пойдет об обработке задач в асинхронном режиме. Из этой главы вы узнаете, как настроить и задействовать Celery для обработки длительных процессов таким образом, чтобы не мешать основному потоку приложения, делая его работу более динамичной. В продолжение нашего исследования брокеров сообщений мы представим RabbitMQ как еще один хороший выбор для управления асинхронными задачами. Чтобы помочь читателям понять уникальные преимущества RabbitMQ и его соответствие различным сценариям, в главе проводится сравнение его функций и возможностей с другими брокерами.

Подводя итог, можно сказать, что из этой главы читатели почерпнут знания, необходимые для создания эффективных и масштабируемых систем, научатся объединять и внедрять методы асинхронной обработки задач и посредничество брокеров сообщений в передаче сообщений между приложениями.

Обзор брокеров сообщений

Брокеры сообщений занимают ключевое место в современных архитектурах внутренних систем, обеспечивая эффективное взаимодействие между различными частями приложения, особенно в распределенных системах. Эти приложения управляют сообщениями между отправителями и получателями сервисов, обеспечивая надежную передачу данных без непосредственного соединения компонентов, что способствует развитию модульной архитектуры.

Роль брокеров сообщений во внутренних приложениях

Брокеры сообщений — это промежуточное программное обеспечение, которое занимается передачей сообщений между различными компонентами приложения. Брокеры сообщений занимают важное место в сценариях с архитектурой микросервисов, где сервисы должны взаимодействовать асинхронно и оставаться слабосвязанными между собой.

Перечислим основные функции брокера сообщений.

- *Разделение компонентов приложения.* Брокеры сообщений позволяют различным частям приложения взаимодействовать, не будучи напрямую связанными, что уменьшает их зависимость друг от друга и облегчает масштабирование и обслуживание.
- *Асинхронная связь.* Брокеры сообщений обеспечивают асинхронную обработку, позволяя компонентам ставить сообщения в очередь, не дожидаясь ответа, что повышает скорость реакции и эффективность приложения.
- *Балансировка нагрузки.* Брокеры могут распределять задачи или сообщения между несколькими потребителями таким образом, чтобы оптимизировать использование ресурсов и повысить общую производительность системы.
- *Отказоустойчивость.* Разделяя сервисы, брокеры сообщений помогают отделить сбои, возникающие в одной части системы, от других частей. Брокеры часто обеспечивают сохранение сообщений, гарантируя, что сообщения не будут потеряны в случае сбоев в обработке.
- *Гарантированная доставка.* Большинство брокеров сообщений поддерживают механизмы, которые гарантируют, что сообщения не будут потеряны — они будут доставлены как минимум один раз или точно один раз, что очень важно для критически важных данных транзакций.

Redis как брокер сообщений

Redis общеизвестен как хранилище данных в памяти. Redis поддерживает различные структуры данных, такие как списки, наборы и сортированные наборы. Эти наборы данных могут быть использованы для реализации очередей доставки сообщений. Поэтому Redis эффективно работает в качестве брокера сообщений.

Каким образом Redis обеспечивает обмен сообщениями?

Redis предоставляет два основных механизма для работы с сообщениями: Pub/Sub (Publish/Subscribe или публикация/подписка) и

постоянные очереди. Хотя Pub/Sub в Redis не гарантирует сохранность сообщений (если сообщение опубликовано, а подписчиков нет, сообщение будет потеряно), он является невероятно быстрым и простым для сценариев, где потери сообщений не являются критичными.

Ниже приведен пример использования Redis в Python:

```
import redis
# Подключение к Redis
r = redis.Redis(host='localhost', port=6379, db=0)
# Публикация
r.publish('emailChannel', 'Send welcome
                                        email to user@gitforgits.com')
# Подписчик
pubsub = r.pubsub()
pubsub.subscribe('emailChannel')
for message in pubsub.listen():
print(сообщение)
```

Использование списков в качестве очередей:

- Передача сообщений — компоненты могут отправлять сообщения в список Redis, который используется для создания очереди.
- Выборка сообщений — компоненты могут извлекать сообщения из списка для обработки.

Ниже приведен пример:

```
# Производитель
r.lpush('taskQueue', 'task1')
# Потребитель
task = r.rpop('taskQueue')
if task:
print(f "Обработка {task.decode(„utf-8")}")
```

В нашем университетском приложении Redis можно использовать в качестве брокера сообщений для решения таких задач, как отправка уведомлений студентам, запись студентов на курсы. Также Redis может выполнять межсервисное взаимодействие в архитектуре микросервисов. Например, когда новый курс добавляется через службу управления курсами, эта служба может

опубликовать сообщение в канал Redis, на который подписана служба уведомлений, чтобы оповестить студентов о доступности нового курса.

Интеграция Kafka для обработки данных в реальном времени

Описание Apache Kafka

Система Kafka работает в кластере из одного или нескольких серверов (брокеров) и упорядочивает сообщения в темы. Каждое сообщение в теме состоит из ключа, значения и временно́й метки. Kafka отличается высокой устойчивостью к сбоям узлов и поддерживает автоматическое восстановление данных. Ее производительность не сильно снижается при увеличении объема передаваемых данных, что делает ее идеальной службой для работы с приложениями с огромными потоками данных, таких как телеметрия или протоколирование данных из нескольких источников.

> ***Примечание переводчика***
>
> Apache Kafka — это распределенная система, предназначенная для обработки потоков данных в режиме реального времени. Ее можно сравнить с почтой: одни сервисы передают в систему сообщения-письма, а другие — получают. Apache Kafka называют брокером сообщений, потому что эта служба выступает в качестве посредника.

Основные компоненты Kafka

- *Производитель.* Любое приложение, которое публикует данные (пишет) в темы Kafka.
- *Брокер.* Сервер в кластере Kafka, который хранит данные и обслуживает клиентов.
- *Тема.* Категория или лента, в которую публикуются записи. Темы в Kafka являются многопользовательскими: у них может быть ни одного, один или много потребителей, которые подписываются на публикуемые данные.

- *Потребитель.* Приложения или процессы, которые подписываются на темы (читают их) и обрабатывают поток опубликованных сообщений.
- *Zookeeper.* Управляет и координирует работу брокеров Kafka. Он необходим для управления состоянием кластера.

Интеграция Kafka для обработки данных в режиме реального времени

Сейчас мы разберем, какие нужно выполнить шаги по настройке Kafka, чтобы университетское приложение могло обрабатывать запросы о регистрации и динамически обновлять информацию о наличии курсов в режиме реального времени.

Установка и настройка Kafka

Начните с загрузки установочного файла Kafka с официального сайта Apache Kafka и распакуйте его. Kafka использует службу Zookeeper для управления и координации брокеров. Запустите Zookeeper с помощью следующей команды:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

После того как служба Zookeeper будет запущена, запустите сервер Kafka:

```
bin/kafka-server-start.sh config/server.properties
```

Создание тем

Создайте тему с именем `course-enrollment` (зачисление на курс), которую ваше приложение будет использовать для обработки запросов на зачисление.

```
bin/kafka-topics.sh --create --topic course-enrollment --bootstrap-
server localhost:9092 --replication-factor 1 --partitions 1
```

Реализация производителей и потребителей Kafka

Предположим, что нам нужно обрабатывать записи на курсы в режиме реального времени и отправлять уведомления.

Код производителя. Эта часть приложения отправляет сообщения в Kafka всякий раз, когда студент записывается на курс.

```
from kafka import KafkaProducer
import json
producer = KafkaProducer(bootstrap_servers= ['localhost:9092'],
value_serializer=lambda v: json.dumps(v).encode('utf-8'))
def send_enrollment(enrollment_info):
producer.send('course-enrollment', enrollment_info)
producer.flush()
send_enrollment({'student_id': 123, 'course_id': 'CS101'})
```

Код потребителя. Эта часть считывает сообщения о регистрации и обрабатывает их, например обновляет базу данных или запускает выполнение других действий.

```
from kafka import KafkaConsumer
import json
consumer = KafkaConsumer('course-enrollment',
bootstrap_servers=['localhost:9092'],
auto_offset_reset='earliest',
value_deserializer=lambda x:json.loads(x.decode('utf-8')))
for message in consumer:
enrollment_info = message.value
print(f"Received enrollment info: {enrollment_info}")
# Обработка регистрации, например обновление базы данных,
# отправка уведомлений и т. д.
```

Для ситуаций, когда требуется минимальная задержка, но необходимо обеспечить высокую пропускную способность, например при обработке данных о зачислении студентов или обновлении информации о курсах на разных факультетах, Kafka — это то, что нужно, благодаря своей масштабируемости и способности работать с огромными объемами данных. Сценарии для производителей и потребителей позволяют обрабатывать данные асинхронно, что делает приложение быстро реагирующим на поступающие запросы и эффективным.

Асинхронная обработка задач с помощью Celery

Знакомство с асинхронной обработкой задач

В процессе асинхронной обработки задач происходит одновременное выполнение не только этих задач, но и всех процессов приложения. Как правило, задачи и процессы выполняются в отдельных потоках, чтобы основное приложение могло продолжать свою работу без сбоев. Асинхронная обработка задач — важнейшая техника в современных приложениях, позволяющая выполнять длительные или ресурсоемкие операции в фоновом режиме, не блокируя основной поток приложения. Этот метод особенно полезен для таких операций, как пакетная отправка электронной почты, обработка файлов или создание отчетов, которые при синхронном выполнении могут занимать значительное количество времени и отрицательно сказываться на работе пользователей.

Использование Celery для асинхронной обработки задач

Celery — это система, основанная на распределенной передаче сообщений, представляющая собой мощную и гибкую асинхронную очередь выполнения задач и процессов. Система ориентирована на работу в реальном времени, но поддерживает и работу в режиме расписания. Единицы выполнения, называемые заданиями, выполняются параллельно на одном или нескольких рабочих узлах с использованием многопроцессорности, библиотек Eventlet или Gevent. Общение в Celery происходит через сообщения, обычно с использованием брокера сообщений в качестве посредника между клиентами и рабочими узлами.

> ***Примечание переводчика***
>
> Eventlet — это библиотека для асинхронного программирования и параллельного выполнения задач в Python.
>
> Gevent — это высокопроизводительная асинхронная сетевая библиотека для Python. Эта библиотека специально разработана для решения проблем с параллельными вычислениями за счет ис-

пользования корутинов, известных как гринлеты. Gevent предлагает оптимизированный подход к асинхронному программированию, упрощая процесс разработки и способствуя эффективной многозадачности.

Как работает Celery?

Мы уже рассказывали о Celery в одной из предыдущих глав, но сейчас еще раз кратко разберем его ключевые компоненты и принцип работы в целом.

Celery взаимодействует с очередью заданий через брокер сообщений, такой как RabbitMQ или Redis. Брокер сообщений содержит очередь заданий, которые должны быть выполнены. Рабочие процессы постоянно следят за брокером сообщений и выполняют задания по мере их появления в очереди. Celery может хранить или отправлять результаты выполнения заданий с помощью внутреннего сервера, например Redis, MongoDB или AMQP.

Использование Celery

Установка Celery и Redis

Для начала убедитесь, что Celery и Redis установлены.

```
pip install celery redis
```

Настройка Celery

Создайте новый файл **celery_app.py** в каталоге вашего университетского приложения. В этом файле будут храниться настройки вашего приложения Celery.

```
from celery import Celery
app = Celery('university_tasks',
broker='redis://localhost:6379/0',
backend='redis://localhost:6379/0')
@app.task
def send_email(student_email, message):
# Имитация отправки электронного письма
print(f"Отправка письма на {student_email} с сообщением: {message}")
return 'Email sent'
```

```
@app.task
def process_enrollment(course_id, student_id):
# Имитация процесса зачисления
print(f "Обработка записи студента {student_id} на курс {course_id}")
return 'Зачисление завершено'
```

Запуск Celery Worker

Чтобы приступить к выполнению задач, необходимо предварительно запустить рабочий процесс Celery (Celery Worker) из каталога проекта.

```
celery -A celery_app worker --loglevel=info
```

Эта команда запускает рабочий процесс Celery Worker, который прослушивает задания в очереди Redis и обрабатывает их по мере поступления.

Постановка задач в очередь

Теперь из другой части вашего приложения или из отдельного скрипта (сценария) вы можете поставить задачи в очередь на обработку с помощью этого процесса.

```
from celery_app import send_email, process_enrollment
# Постановка в очередь задачи на отправку письма
send_email.delay('student@gitforgits.com', 'Добро пожаловать на курс!')
# Поставить в очередь задачу на обработку регистрации
process_enrollment.delay('CS101', '12345')
```

Метод `.delay()` используется для отправки задач брокеру в асинхронном режиме, которые затем перехватываются процессом Celery Worker. При такой конфигурации выполнение таких сложных операций, как отправка уведомлений по электронной почте или зачисление студентов, может происходить в фоновом режиме без замедления работы приложения для конечных пользователей.

RabbitMQ как альтернативный брокер сообщений

Брокеры сообщений, используемые в микросервисах или архитектурах с событийно-ориентированной парадигмой, необходимы

для управления обменом данными и связью между компонентами распределенных систем. Благодаря тому, что брокеры сообщений обеспечивают формирование очереди сообщений, их доставку и обработку, а также позволяют разделить компоненты приложения, системы легче масштабировать и поддерживать. Хотя Kafka и Redis — хорошо известные варианты, RabbitMQ — не плохая альтернатива с широкими возможностями обмена сообщениями.

Описание RabbitMQ и принцип его работы

RabbitMQ — это брокер сообщений с открытым исходным кодом, который известен своей надежностью, гибкостью и поддержкой множества протоколов обмена сообщениями, включая AMQP (Advanced Message Queuing Protocol или расширенный протокол очереди сообщений). Этот брокер предлагает множество функций, таких как постановка сообщений в очередь, подтверждение доставки, гибкая маршрутизация и транзакции, что делает его хорошим выбором для работы в сложных приложениях, требующих надежных механизмов доставки сообщений, таких как системы обработки заказов, банковские операции или системы обработки данных в реальном времени.

Необходимость использования альтернативного брокера сообщений, такого как RabbitMQ, в архитектуре внутренней части любого приложения обусловлена потребностью в высокой пропускной способности и надежной доставке сообщений в распределенной среде. RabbitMQ предоставляет эти возможности и гарантирует, что сообщения не будут потеряны даже в случае сбоев в работе потребителей, что очень важно для критически важных бизнес-процессов.

Интеграция RabbitMQ

А что если предположить, что нашему университетскому приложению требуется надежная обработка сообщений для таких событий, как уведомления о регистрации, обновления информации о курсах или даже передача административных команд, которые должны надежно распространяться по различным частям системы.

Установка RabbitMQ

Во-первых, убедитесь, что брокер сообщений RabbitMQ установлен в вашей системе. RabbitMQ может поддерживаться различными операционными системами, а также может быть установлен в виде образа Docker.

Для базовой установки на Ubuntu используйте следующие команды:

```
sudo apt update
sudo apt install rabbitmq-server
sudo systemctl enable rabbitmq-server
sudo systemctl start rabbitmq-server
```

Затем для удобства работы можно включить интерфейс управления:

```
sudo rabbitmq-plugins enable rabbitmq_management
```

Настройка RabbitMQ в приложении

После запуска RabbitMQ необходимо интегрировать его в университетское приложение с помощью Celery, который мы ранее настроили для работы с Redis.

Обновите файл **celery_app.py**, чтобы использовать RabbitMQ в качестве брокера. Измените URL-адрес брокера, чтобы использовать RabbitMQ:

```
from celery import Celery
app = Celery('university_tasks',
broker='amqp://user:password@localhost/',
backend='rpc://')
@app.task
def send_notification(message):
print(f"Notification: {message}")
return "Notification sent"
@app.task
def process_registration(details):
print(f"Registration details: {details}")
return "Registration processed"
```

В приведенном выше фрагменте кода `amqp://user:password@localhost/` — это строка подключения для RabbitMQ по умолчанию. Замените `user` и `password` на соответствующие учетные данные.

Запустите рабочий процесс Celery, который будет прослушивать сообщения от RabbitMQ:

```
celery -A celery_app worker --loglevel=info
```

Отправка сообщений

Настроив Celery и RabbitMQ, вы можете отправлять задания в очередь из любой точки вашего приложения:

```
from celery_app import send_notification, process_registration
# Определить задачи
send_notification.delay("Доступен новый курс!")
process_registration.delay("Студент Джон Доу зарегистрировался
на курс „Физика 101"").
```

Когда в сложных системах требуются расширенные возможности маршрутизации и гарантии доставки сообщений, RabbitMQ становится оптимальным выбором благодаря своей надежности.

Резюме

В этой главе мы рассмотрели особенности работы брокеров сообщений и асинхронной обработки задач, разделив их роли в повышении производительности внутренних приложений. Это особенно актуально, когда речь идет о сложных операциях, требующих масштабируемости и надежности. В первом разделе главы было дано исчерпывающее представление о брокерах сообщений и описана их важнейшая функция в обеспечении надежной связи между компонентами распределенных приложений. Общая эффективность и надежность приложения повышаются благодаря такой настройке, которая разделяет части системы и оптимизирует поток данных между процессами.

Брокер сообщений Apache Kafka, обладающий преимуществами в обработке данных в реальном времени, стал следующей темой

для изучения в этой главе. Учитывая его способность управлять высокопроизводительными потоками данных, Kafka хорошо подходит для ситуаций, требующих быстрой обработки и оперативного реагирования, например для работы университетской системы с транзакционными данными в режиме реального времени.

Кроме того, в этой главе был описан сервис Celery, представляющий собой очередь задач. Этот сервис взаимодействует с RabbitMQ и Redis, двумя брокерами сообщений, и эффективно справляется с фоновыми задачами. Университетское приложение смогло сохранить скорость отклика пользовательского интерфейса благодаря интеграции Celery, который снял нагрузку с ресурсоемких задач, таких как массовая отправка электронных писем или асинхронная обработка запросов пользовательских данных.

В заключение мы перешли к изучению RabbitMQ, альтернативного брокера сообщений с мощными функциями создания очереди сообщений. Критически важные приложения, которые не могут позволить себе потерю сообщений, оценят его устойчивость и надежность в работе с сообщениями. Интеграция RabbitMQ в университетское приложение улучшила коммуникационные возможности и возможности управления обработкой задач.

Эпилог

Я надеюсь, что после прочтения данной книги вы стали лучше разбираться в предмете и получили инструменты, необходимые для того, чтобы с головой окунуться в мир разработки внутренних компонентов, написанных на языке Python. Путешествие по этой книге призвано вдохновить вас на осознанный подход к разработке надежного, масштабируемого и безопасного программного обеспечения в дополнение к изучению технических аспектов и инструментов, которые делают внутренние системы эффективными.

Мы рассмотрели все, начиная с основ программирования на Python и заканчивая более сложными темами, такими как интеграция с базами данных, асинхронная обработка и безопасность. Мы также познакомились с веб-фреймворками Flask и FastAPI. Комплексный подход к обучению, отражающий на примерах реальный ход развития проектов по разработке программного обеспечения, был достигнут благодаря тщательному построению каждого раздела на основе предыдущего. Задача при рассмотрении этих примеров и проектов заключалась в том, чтобы дать вам реальный опыт и информацию, которые вы сможете использовать при разработке своих собственных проектов. Мы хотели показать вам все тонкости проектирования внутреннего кода, от создания среды разработки до развертывания приложений в облаке, чтобы вы были готовы к принятию решений и преодолению трудностей, с которыми столкнетесь в процессе работы.

Не забывайте, что технология — это динамично развивающаяся область, которая постоянно меняется по мере вашего развития. Несмотря на то что рассмотренные нами методы и инструменты являются самыми современными, постоянное стремление к зна-

ниям необходимо каждому успешному разработчику. Я надеюсь, что вы будете продолжать искать новую информацию, пробовать различные технологии и совершенствовать свои способности.

Принимайте участие в работе сообществ, вносите свой вклад в проекты с открытым исходным кодом и увековечивайте цикл обмена знаниями. Ваши знания и навыки как разработчика будут расширены благодаря участию в этих мероприятиях. Люди, с которыми вы познакомитесь, будут разделять ваши интересы и обеспечат вам дружеское общение, совет и поддержку.

Я надеюсь, что из этой книги вы почерпнули самое важное — как стратегически подходить к разработке проектов внутреннего кода. Программирование — это, прежде всего, поиск решений проблем, каким бы важным ни было написание кода. Каждая написанная вами строчка вносит свой вклад в нечто большее, будь то внутренний инструмент с несколькими тысячами пользователей или система с миллионами пользователей.

И последнее, что я хочу сказать, но не менее важное. Я хочу поблагодарить вас за то, что вы выбрали эту книгу в качестве своего пути к знаниям и за вашу преданность обучению. Делиться с вами тем, что я узнал, было для меня большой честью. Хотя стать экспертом в области разработки внутреннего кода не так-то просто, результат стоит затраченных усилий. Никогда не прекращайте учиться, никогда не переставайте заставлять себя и, самое главное, никогда не переставайте наслаждаться творческим и инновационным процессом.

ПРОГРАММИРОВАНИЕ БЭКЕНДА НА Python

Практическое руководство

Эта книга — не просто руководство по программированию, а книга-маршрут, которая поможет выйти на уровень настоящего эксперта по серверной части стека, умеющего проектировать и развертывать мощные и эффективные приложения.

В этой книге:

- Как писать эффективный и хорошо структурированный код на Python, придерживаясь наилучших практик
- Как обеспечить эффективность приложений и упростить их масштабирование, применяя техники асинхронного программирования
- Какова роль Kubernetes и Docker в оркестрации и контейнеризации приложений на Python
- Как эксплуатировать облачные сервисы для обеспечения гарантированно высокой доступности и максимальной производительности
- Как усовершенствовать обработку данных путем интеграции с базами данных при помощи SQLAlchemy.
- Как защищать веб-приложения, настраивая механизмы авторизации и аутентификации при помощи OAuth
- Как эффективно обрабатывать данные в режиме реального времени при помощи брокеров сообщений RabbitMQ и Kafka
- Как сократить количество ошибок, реализовать непрерывную интеграцию и непрерывное развертывание

ISBN 978-5-9775-2096-6

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

GitforGits
ASIAN PUBLISHING HOUSE

bhv®